synopsys®

DesignWare® Cores Driver Software Development Kit (SDK) User Guide

For use with the following DesignWare® Cores

DWC SPAcc — Product Code: B259-0

DWC SPAcc LTE — Product Code: B261-0

DWC SPAcc QoS — Product Code: B837-0

DWC SPAcc SHA3 — Product Code: B836-0

DWC TRNG — Product Code: B226-0

DWC TRNG for ESMs — Product Code: B230-0

DWC TRNG NIST SP800-90C — Product Code: B227-0

DWC ECC/RSA 32bit ALU PKA — Product Code: B231-0

DWC ECC/RSA 64bit ALU PKA — Product Code: B832-0

DWC ECC/RSA 128bit ALU PKA — Product Code: B233-0

Version 1.53
March 2018

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Revision History

Version	Date	Description
1.53	March 2018	Updated: ■ Tables 3-1 and 3-2. Removed: ■ Context Manager sections.
1.52	January 2018	Updated: ■ Section 2.1: Configure the SDK.
1.51	November 2017	Removed: ■ References to obsoleted products.
1.50	April 2016	Updated: ■ DWC TRNG NIST SP800-90C Removed: ■ CLP-27 Section and references. ■ CLP-850 Section and references.
1.40	January 2016	-

1 Overview

The DriverSDK package is reference software with example source code demonstrating how to use various Synopsys Security IP cores. It is implemented and tested for 32-bit Linux using device drivers and user-space samples, along with additional examples, which can be used on non-Linux (that is, bare metal) systems. This software is intended to demonstrate the functionality of the core and includes example plugins for a limited number of cryptographic features in the Linux kernel.

This package is developed as a generic universal driver and includes components/modules, which may not be present in your IP delivery. Each module is designed to operate standalone, comes with a reference Linux kernel module to demonstrate the IP running in a Linux system, and is provided as-is without maintenance or support.

1.1 Driver SDK Package

There are modules for each piece of Synopsys hardware, all of which have a hierarchy that depends on how they are instantiated:

- PDU wrapper (provides basic IO to devices, DDT support)
  - SPAcc device – DWC SPAcc
  - TRNG device – DWC TRNG or DWC TRNG NIST SP800-90C
  - PKA device – DWC PKA

The resources are managed with one of the example PCI or memory based device modules. Located in the *src/pdu/linux/kernel* directory, the following files can be used as templates for creating a platform specific device module.

- spacc_pci.c: Provides ability to bring up a SPAcc.
- spacc_mem.c: Similar to the PCI module but for direct memory mapped platforms

The device modules are what tell the kernel the resources of the devices. The device drivers themselves then bind to this information to provide an interface to the hardware. In this way, the device drivers are platform independent.

1.2 List of Acroynms

Acroynm	Description
AIS	Application Notes and Interpretation of the Scheme
CLP800	Refers to DWC TRNG
CLP890	Refers to DWC TRNG NISP SP800-90C
DF	Derivation Function
DRBG	Deterministic Random Bit Generator
HW	Hardware
KAT	Known Answer Test
NIST	National Institute of Standards and Technology
RNG	Random Number Generator
SW	Software
TRNG	True Random Number Generator
TRNG3	Internal project code name for DWC TRNG. References to TRNG3 are equivalent to DWC TRNG

2 Building the SDK

2.1 Configure the SDK

As shipped the SDK must be configured before it is compiled, otherwise a warning appears indicating that the configuration is required. Users must edit the provided *synopsys/driver/Makefile* file by performing the following steps:

1. Comment out the **CONFIG_ME** definition.
2. Change the lines with **PDU_BASE_ADDR** and **PDU_BASE_IRQ** to reflect the base memory address and IRQ of the SPAcc/PKA/TRNG in your system.
3. Enable module to build by uncommenting one of the following lines.

   #ENABLE_SPACC=1

   #ENABLE_TRNG=1

   #ENABLE_PKA=1

4. For TRNG or PKA, uncomment **ENABLE_SINGLE_CORE=1** and select any one type of core from the following list.

   #ENABLE_SINGLE_CLP800=1 ➔ enable for DWC TRNG

   #ENABLE_SINGLE_CLP890=1 ➔ enable for DWC TRNG NIST SP800-90C

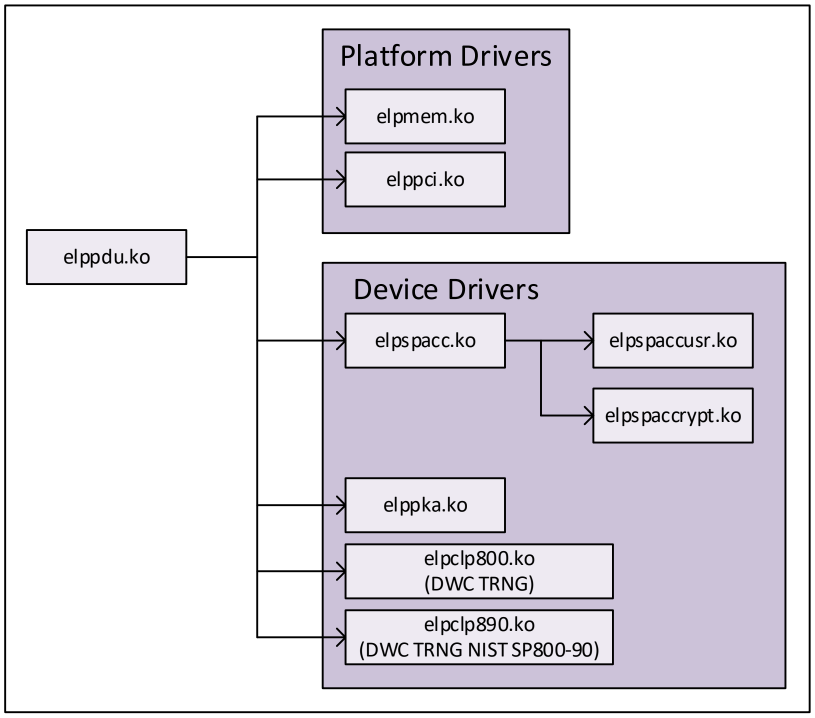
   #ENABLE_SINGLE_PKA=1 ➔ enable for DWC PKA

All other defines can be left disabled.

2.2 Kernel Modules

The build process produces a series of kernel modules listed below:

Figure 2-1 Kernal Modules



At the heart of the system is the *elppdu.ko* module, which provides for the generic memory mapped I/O, DDT, locking, and configuration functionality required by the rest of the drivers.

From this the *platform drivers* are built. These drivers instruct the Linux kernel that devices are present at given memory locations with specified IRQ lines. The *elpmem.ko* module is a generic memory based platform driver, which uses the environment variables **PDU_BASE_ADDR** and **PDU_BASE_IRQ** to specify to the kernel, where the devices are. The *elppci.ko* is a PCI-based platform driver that looks for a device with PCI ID E117:59AE to determine the memory address and IRQ. Both modules assume all cores are tied to the same IRQ and are capable of enumerating a SPAcc.

The rest of the modules are the *device* or *service* drivers, which either add the ability to control and manage the hardware to the Kernel (in the case of device drivers) or add functionality for the user (in the case of service drivers).

- **elpspacc.ko**: This module provides access to the SPAcc, SPAcc-PDU, and SPAcc-HSM devices to other kernel modules. It handles low-level slave register access, managing contexts (allocating/storing/loadings), responding to IRQs, and issuing callbacks to the user.
  - **elpspaccusr.ko**: This module provides the */dev/spaccusr* device, which userspace tasks can use to perform cipher/hash jobs.
  - **elpspacccrypt.ko**: This module provides for Linux Kernel CryptoAPI support combined AEAD and individual hash jobs to process IPsec ESP and AH datagrams in the kernel. It is not intended to be a general purpose CryptoAPI driver. The driver supports all of the modes of the SPAcc (with respect to approved IPsec modes) with the following exceptions:
  - Ciphersuites **not** yet supported:

    - AH-GMAC (Linux Kernel does not support this mode as of 3.11.5)
    - ESP-CCM

- elpclp800.ko, elpclp890.ko: These modules provides access to the DWC TRNG and DWC TRNG NIST SP800-90C (resp.) TRNG devices for random number generation. The modules handle responding to IRQs and programming the low-level slave ports. It will also export the TRNG functionality to the Kernel, which will show up to the user as /dev/hwrng, and can be read for random bits from the engine.
- **elppka.ko**: This module provides access to the PKA for public key acceleration.

2.2.1 Kernel sysfs Entries

The DWC SPAcc, DWC PKA, DWC TRNG, and DWC TRNG NIST SP800-90C drivers all support sysfs access to the HW core in some fashion or another. In the case, of the SPAcc core, the file */sys/devices/platform/spacc/reg* can be read to display the current status of the slave port registers for their respective core. The SPAcc file can be written to in the form "EPN,virtual". For example, "0x1234,0" to specify, which device to examine and which virtual SPAcc.

All of these files are accessible only as root, where security is a concern. Non-root users will be denied access to the files.

2.3 Return Codes

All functions that return error codes return an *int*. A negative value indicates error, zero indicates success. Some functions (such as context and job allocation) return a positive integer to indicate the index of the context allocated.

2.4 Application Provided Functions

Some of the cores require functions to be provided to the SDK by the host application. These are for platform specific reasons to allow the SDK greater portability. The list of required functions are**:**

- PDU_INIT_LOCK()
- PDU_LOCK()
- PDU_UNLOCK()
- pdu_mem_init()
- pdu_mem_deinit()
- pdu_ddt_init()
- pdu_ddt_add()
- pdu_ddt_reset()
- pdu_ddt_free()
- pdu_io_read32()
- pdu_io_write32()
- pdu_malloc()
- pdu_free()
- pdu_dma_alloc()

- pdu_dma_free()
- pdu_sync_single_for_cpu()
- pdu_sync_single_for_device()
- pdu_get_version()
- pdu_error_code()

2.4.1 Porting Outline

The core of the SDKs (such as, src/core/spacc/, and so on) are all relatively portable and dependent, the non-portable platform specific code is placed in the respective Kernel directories. These are Linux instantiations of the SDK in a live model (for example, the devices are accessible on real hardware to the programmer).

A developer porting to a non-Linux platform will want to modify the following two files first.

- **src/pdu/bare/include/elppdu.h**: This file needs to be updated with platform specific types and functions for operations, such as locking and printing messages. The developer will want to remove any non-supported header include statements.
- **src/pdu/bare/pdu/pdu.c:** This file has non-portable code at the top half for tasks such as memory based I/O reading and writing, allocating DDT structures, allocating virtual memory, and so on.

Once those two are updated each of the individual "core" SDK directories should be useable with a few additions. Users will want to link the contents of *src/pdu/bare/pdu/libpdu.a* with the core SDK they are using to ensure the pdu functions are available.

To build the SDK libraries ensure that the definition of **PDU_USE_KERNEL** is undefined or removed in the top-level makefile. This will build the files under *src/pdu/bare/* and also force the individual device SDKs to build as libraries instead of kernel modules. After running "make" all of the library files will be available under *bin/*.

Building the bare libraries will give the developer access to control the hardware but work is required to put it in a live environment. The Linux kernel modules include additional code such as, IRQ handlers and initialization that cannot be part of a portable library.

After writing the PDU functions required under *src/pdu/bare/pdu/pdu.c*, the developer will next need to:

1. Call the SDK init function (such as spacc_init()) with the appropriate base memory address of the HW registers. This may need to be memory mapped into the virtual address space, if the system uses an MMU. Otherwise, it may be the physical address of the hardware.
2. Provide and register an IRQ handler for the OS to handle device interrupts.
3. Call the SDK function provided to enable and configure interrupts.
4. Integrate calls to the SDK into whatever stack (for example, IPsec) being used.

2.4.2 Macros

The following macros are found in *synopsys/driver/src/pdu/linux/include/elppdu.h*.

```
#define PDU_MAX_DDT 64
```

This denotes how many entries a single DDT can contain. By default, there are at most 64 (65 including the terminator entry), which requires 520 bytes of memory. The Linux version of the PDU functions use a DMA pool internally to help cut down on the memory usage. This value should not be set below 16 and generally no higher than 64.

```
#define PDU_DMA_ADDR_T dma_addr_t
```

This is the type of address used for DMA addresses. Typically, it is merely a 32-bit value but in Linux kernel modules it is *dma_addr_t* and can change size even on 32-bit platforms (for instance, on x86 with PAE enabled).

The PDU implementation must provide functionality for locking to prevent race conditions in the case that multiple calling threads are trying to access the SDKs in parallel. It must define a lock data type, for instance, with Linux it is:

```
#define PDU_LOCK_TYPE spinlock_t
```

Followed by functions, which initialize, lock, and unlock the lock. For instance, with Linux they are:

```
#define PDU_INIT_LOCK(lock) spin_lock_init(lock)
#define PDU_LOCK(lock, flags) spin_lock_irqsave(lock, flags)
#define PDU_UNLOCK(lock, flags) spin_unlock_irqrestore(lock, flags,
```

where *flags* is an *unsigned long* variable that is presumably written to by the LOCK and read by UNLOCK. If it is not needed in a particular platform it can be ignored.

Note that the SDKs will pass the address of whatever type specified by PDU_LOCK_TYPE to the functions. If they do not expect a pointer, the macro should dereference the pointer passed by the SDK. While these macros are called by the SDK in a general sense, they might be part of an interrupt handler. They should map to functions suitable for interrupt handlers.

For non-interrupt based locks there are

```
#define PDU_LOCK_BH(lock,flags) spin_lock(lock); (void)flags
#define PDU_UNLOCK_BH(lock,flags) spin_unlock(lock); (void)flags
```

These macros are not meant for interrupt handlers.

2.4.3 DDT Support

The PDU implementation is platform specific but it must have this structure (in *src/pdu/include/elppdu.h*)

```
typedef struct {
   PDU_DMA_ADDR_T phys;
   unsigned long *virt;
   unsigned long idx, limit, len;
} pdu_ddt;
```

The *phys* and *virt* pointers map to the **SAME** 32-bit array (the former being the physical address) that holds the pair of pointer/length DDT values. The *len* parameter holds the length in bytes of the contents pointed to by the DDT table. This structure is managed by the following four functions.

```
int pdu_ddt_init(pdu_ddt *ddt, unsigned long limit);
```

This function must initialize a DDT array with *limit* elements (not including the **NULL** terminator). It must populate the *phys* and *virt* pointers (corresponding to each other) as well as set the *len* to zero. Returns zero if the table has been initialized. In the Linux implementation the 31st bit of *limit* is used to indicate whether the driver should use **GFP_ATOMIC** or **GFP_KERNEL** for allocating memory.

```
int pdu_ddt_add(pdu_ddt *ddt, PDU_DMA_ADDR_T phys, unsigned long size);
```

This function must add a known physical address pointed to by *phys* of length *size* bytes to the DDT array, and increment the *len* parameter. The caller must perform any virtual to physical mapping required. Returns zero if the entry was added.

```
int pdu_ddt_reset(pdu_ddt *ddt);
```

This function should reset the DDT table (clear the entries and zero the *len* member).

```
int pdu_ddt_free(pdu_ddt *ddt);
```

This function frees the allocated DDT table.

2.4.4 I/O Support

```
void pdu_io_write32(void *addr, unsigned long val);
```

This function writes a 32-bit word *val* to the address pointed to by *addr*. The address is assumed to be a virtual address (or something the CPU can write to) already mapped to the physical address.

In the Linux kernel, this maps to the *writel()* function.

```
unsigned long pdu_io_read32(void *addr);
```

This function reads a 32-bit word from the address pointed to by *addr*. The address is assumed to be a virtual address.

In the Linux kernel this maps to the *readl()* function.

Note	In the Linux versions of these functions writing to address **NULL** enables debugging support, which then prints traces to the kernel log when the driver reads/writes from memory. The trace can be disabled by writing to address **NULL** a second time.

2.4.5 Heap Memory Support

```
void *pdu_malloc(unsigned long n);
```

This function allocates *n* bytes of memory (does not have to be page contiguous) and returns the pointer, or **NULL** on error.

In the Linux kernel this maps to the *vmalloc()* function. Note that *kmalloc()* could be used but is not optimal since code that calls *pdu_malloc*() does not require the allocated block to be aligned or even contiguous in physical memory.

```
void pdu_free(void *p);
```

This function frees a block of memory allocated with *pdu_malloc()*.

In the Linux kernel this maps to the *vfree()* function.

2.4.6 DMA Support

```
void pdu_sync_single_for_device(uint32_t addr, uint32_t size);
void pdu_sync_single_for_cpu(uint32_t addr, uint32_t size);
```

These functions take a physical address and size and synchronize the data either off CPU (*for_device*) or in CPU (*for_cpu*). In Linux for instance they map to calls to *dma_sync_single_for_*()*. On memory coherent devices these functions can map to empty functions.

```
void *pdu_dma_alloc(size_t bytes, PDU_DMA_ADDR_T *phys);
void *pdu_dma_free(size_t bytes, void *virt, PDU_DMA_ADDR_T phys);
```

These functions allocate and free DMA coherent memory, which does not require manual synchronization calls. It effectively must allocate uncacheable memory. In the Linux kernel this maps to calls to the *dma_alloc_coherent()* functions.

The allocation function must return the *virtual* address (or **NULL** upon error) and store the physical address in *phys*. The free function must be passed with the same size block as the allocation call.

2.4.7 Library Support

```
int pdu_error_code(int err);
```

This function maps the SDK error codes into system-specific error codes. In the Linux kernel this produces results such as -EINVAL, -ENOENT, and so on. It is acceptable for this function to return its argument unchanged.

2.4.8 SPAcc Userspace Profiling

The SPAcc profiling uses the userspace driver to benchmark a realistic path from a userspace application through the kernel and device and back again. The tool is called *spaccdevoff* and is built by default when the SPAcc module is built.

```
spaccdevoff [--size jobsize --cipher ciphermode --ciphersize keysize --hash hashmode --
hashsize keysize --runs count --threads threadcount --inplace]
```

The cipher and hash algorithm choices are the same as in the RE benchmark program. The output style is essentially the same as all of the other tools with the exception that the first column is the number of threads being used.

3 DWC SPAcc

3.1 Initialization

The SPAcc library operates on a *spacc_device* device context, which is mapped to a single SPAcc core (whether a native SPAcc device or a virtual-SPAcc interface).

```
int spacc_init(void *baseaddr, spacc_device *spacc, pdu_info *info);
```

This will initialize the device based on the given base address in virtual memory *baseaddr*. It requires the PDU device information to be filled out in *info* with a prior call *to pdu_get_version()*. After this call there will be allocated memory inside the SPAcc structure, which must be freed with a call to *spacc_fini()*.

3.2 Job Notification

3.2.1 IRQ Support

An IRQ handler for the SPAcc need only call *spacc_process_irq()*. Ideally, the call could take place from a tasklet (a non-hard IRQ context) but ideally should be quick enough to be called from a hard interrupt context. This function call requires that the user initialize the call back functions for the various interrupts (see the description of spacc_process_irq(), later in the document).

The Linux kernel module by default enables a STAT_WD and STAT interrupt and operates in IRQ mode, which means it is primarily driven by IRQs to collect terminated jobs. This is accomplished in the Linux kernel module with the following code:

```
/* register irq callback function */
priv->spacc.irq_cb_stat = spacc_stat_process;
priv->spacc.irq_cb_stat_wd = spacc_stat_process;

/* set threshold and enable irq */
spacc_irq_stat_enable (&priv->spacc, priv->spacc.config.ideal_stat_level);
spacc_irq_stat_wd_enable (&priv->spacc);
spacc_irq_glbl_enable (&priv->spacc);
/* enable the wd */
spacc_set_wd_count(&priv->spacc, 250000); // 1msec @250MHz, adjust accordingly
```

This assigns the same callback function for both types of IRQ, the callback function simply schedules a soft-IRQ (outside the IRQ scope) call to *spacc_pop_packets()*, which dequeues jobs off the STAT FIFO. This initialization code also assigns a STAT CNT value to trigger the STAT IRQ on, when the STAT FIFO is filling up. Ideally, don't wait for the FIFO to fill completely as it will stall the SPAcc but with too value low, won’t get the benefit of a longer FIFO.

Finally, this code initializes the STAT and STAT_WD interrupts and then enables the global interrupts on the SPAcc device.

3.2.2 Watchdog Support

The default IRQ model for the SPAcc SDK is to use the STAT_WD and STAT interrupts to catch high-volume traffic and occasional bursts. The STAT interrupt will fire when the STAT FIFO fills up (programmed with a STAT_CNT just below the maximum) and the STAT_WD interrupt will fire when the device idles for a given amount of time. The delay can be specified with this function.

```
int spacc_set_wd_count(spacc_device *spacc, uint32_t val);
```

This will set the SPAcc watchdog timer to the value specified by *val* it is indicated in core clock cycles. For example, with a 50 MHz device a value of 50,000 indicates 1 millisecond.

As described in the *DWC SPAcc Databook* the watchdog works as follows:

1. A write to the CMD0 FIFO, resets the timer value.
2. Once the STAT FIFO has at least one entry the timer starts running.
3. The timer stops running, when the STAT FIFO empties.

When the timer matches the value programmed it will fire the STAT_WD interrupt. To enable the WD interrupt, the function:

```
void spacc_irq_stat_wd_enable (spacc_device *spacc);
```

Can be called. This turns on the bit in the IRQ_EN register, which allows STAT_WD interrupts to occur. They can be disabled with

```
void spacc_irq_stat_wd_disable (spacc_device *spacc);
```

When the STAT_WD interrupt fires, the user should acknowledge the interrupt and then pull jobs off the STAT FIFO. The value programmed in the timer should be large enough such that the CPU can pull jobs off, otherwise subsequent STAT_WD interrupts will be raised possibly causing a deadlock. A workaround is to disable the STAT_WD interrupt in your IRQ handler and then re-enable it from your soft-IRQ (equivalent) task after calling *spacc_pop_packets()*.

3.2.3 IRQ Management (Linux only)

For Linux users of the SPAcc driver there is a tool *bin/spaccirq*, which allows root users to alter how the SPAcc IRQ system works. The command has the following options:

```
spaccirq: --irq_mode wd|step --epn 0x???? [--virt ??] [--wd nnnn] [--stat nnnn] [--cmd nnnn]
   --irq_mode  wd==watchdog, step==stepping IRQ
   --epn       EPN of SPAcc to change (in hex, e.g. 0x0605)
   --virt      Virtual SPAcc to change (default 0)
   --wd        Watch dog counter (in cycles) good values are typically >15000
   --stat      STAT_CNT IRQ trigger, see the print out from loading elpspacc.ko to
               see the size of the STAT FIFO
   --cmd       CMD_CNT IRQ trigger (for CMD0), see the print out from loading
               elpspacc.ko to see the size of the CMD FIFO
```

All of the options default to zero, which inside the kernel means use the defaults.

The IRQ modes are either “wd”, which means to use the watchdog timer (users should specify --wd) or “step”, which means to use the stepping CMD/STAT IRQ system (users should specify --stat and --cmd). The user **must** specify an --irq_mode option in this command.

The SPAcc device which is used to manipulate is specified by --epn for the EPN and optionally --virt for a specific virtual SPAcc. The user **must** specify an --epn option in this command.

The stepping IRQ method uses the --cmd and --stat options to specify the CMD and STAT trigger levels. The values should be zero or higher and less than the total length of the FIFOs. When users load the *elpspacc.ko* module the kernel log will have detailed information about the FIFO depth of the core.

The watchdog IRQ method uses the --wd option, which specifies the watchdog count. It is measured in cycles and should be high enough to prevent too many IRQs from triggering.

Examples:

- Setting watchdog mode with a count of 50000 cycles
  - bin/spaccirq --epn 0x1234 --irq_mode wd --wd 50000
- Setting stepping mode for reset defaults
  - bin/spaccirq --epn 0x1234 --irq_mode step --cmd 0 --stat 1

3.3 Device Termination

To clean up the memory allocated for a SPAcc device, a call to the *spacc_fini()* function is needed.

```
void spacc_fini(spacc_device *spacc);
```

This will free up memory allocated for context and job management.

3.4 Job Programming

Note	There are example job programming routines in *src/core/kernel/example.c* that demonstrate how to program simple cipher and hash jobs. Consult that file if you need help using the SDK.

Programming a job with the SPAcc SDK always follows these general steps:

1. Open a SPAcc job index with *spacc_open()* passing it, which cipher and hash modes are desired.
2. Calls to *spacc_write_context()* to set the hash and/or cipher context (keys, IV, salt).
   - *optionally*, call *spacc_load_skp()* instead to load a key through the secure key port.
3. Optionally, call *spacc_set_key_exp*() to either initialize the key or pre-compute the decrypt key (see the section for this function, for more details).
4. Call to *spacc_set_operation()* to indicate, which direction (encrypt/decrypt), ICV mode, ICV position method (append, offset, and so on), and whether secure key port is used.
5. Construction of a *pdu_ddt* array for both input and output buffers (see the section about PDU)
6. Call to *spacc_packet_enqueue_ddt()* to push the job onto the command FIFO.
7. An interrupt should be tied to spacc_pop_packets() but that function may be polled as well. The calling application should poll *spacc_packet_dequeue()*.

Steps 1-4 may be omitted for repeated jobs, for example, processing bulk data through CBC mode, or partial processing support.

3.4.1 spacc_open()

The *spacc_open()* function allocates a single context from a given SPAcc device and assigns the cipher and hash modes.

```
int spacc_open (spacc_device *spacc, int enc, int hash, int ctx, int secure_mode,
spacc_callback cb, void *cbdata);
```

Where *enc* is one of the enumerated cipher modes, *hash* is one of the enumerated hash modes, *ctxid* is used for resource allocation strategies (use -1 for a new context or pass a known one), and *secure_mode* indicates whether this job is executing in secure mode or not. Secure jobs use a secure key context and require the key to be loaded through *spacc_load_skp()*.

The callback *cb* is called when the job is popped off the status FIFO. It may be set to **NULL** to indicate no callback is desired. The form of the callback is:

```
typedef void (*spacc_callback)(void *spacc_dev, void *data);
```

where *spacc_dev* is the *spacc_device* passed by the caller and *cbdata* is passed as *data*. This callback mechanism allows a user thread waiting on the job to be unblocked.

Note	This callback may be called from an interrupt context and must be IRQ safe.

The allocated job is setup by default to run in complete packet mode, that is, the message begin and end flags will be set. In order to run jobs in partial processing mode the caller must clear these flags and set them as appropriate. See *src/core/kernel/spaccdiag.c,* for an example FSM that executes partial processing mode.

The supported cipher modes are as follows:

Table 3-1 SPAcc Encryption Modes

Encryption Enumeration	Mode Description
CRYPTO_MODE_CHACHA20_STREAM	CHACHA20 mode
CRYPTO_MODE_CHACHA20_POLY1305	CHACHA20/POLY1305 AEAD mode
CRYPTO_MODE_NULL	NULL cipher
CRYPTO_MODE_RC4_40	RC4 40-bit key
CRYPTO_MODE_RC4_128	RC4 128-bit key
CRYPTO_MODE_RC4_KS	RC4 and the RC4 state is manually loaded in the RC4 key context (this allows non-standard key size support)
CRYPTO_MODE_AES_ECB	AES ECB mode
CRYPTO_MODE_AES_CBC	AES CBC mode
CRYPTO_MODE_AES_CTR	AES CTR mode
CRYPTO_MODE_AES_CCM	AES CCM mode

Encryption Enumeration	Mode Description
CRYPTO_MODE_AES_GCM	AES GCM mode (96-bit IV)
CRYPTO_MODE_AES_F8	AES F8 mode
CRYPTO_MODE_AES_XTS	AES XTS mode (with or without CTS)
CRYPTO_MODE_AES_CFB	AES CFB (s=128) mode
CRYPTO_MODE_AES_OFB	AES OFB (s=128) mode
CRYPTO_MODE_AES_CS1	NIST CBC-CS mode 1
CRYPTO_MODE_AES_CS2	NIST CBC-CS mode 2
CRYPTO_MODE_AES_CS3	NIST CBC-CS mode 3
CRYPTO_MODE_MULTI2_ECB	MULTI2 ECB mode
CRYPTO_MODE_MULTI2_CBC	MULTI2 CBC mode
CRYPTO_MODE_MULTI2_OFB	MULTI2 OFB mode
CRYPTO_MODE_MULTI2_CFB	MULTI2 CFB mode
CRYPTO_MODE_3DES_CBC	Triple DES CBC mode
CRYPTO_MODE_3DES_ECB	Triple DES ECB mode
CRYPTO_MODE_DES_CBC	DES CBC mode
CRYPTO_MODE_DES_ECB	DES ECB mode
CRYPTO_MODE_KASUMI_ECB	KASUMI ECB mode
CRYPTO_MODE_KASUMI_F8	KASUMI F8 mode
CRYPTO_MODE_SNOW3G_UEA2	SNOW3G UEA2 encryption mode
CRYPTO_MODE_ZUC_UEA3	ZUC UEA3 encryption mode

The supported hash modes are as follows:

Table 3-2 SPAcc Hash Modes

Hash Enumeration	Mode Description
CRYPTO_MODE_HASH_MD5	MD5 hash mode
CRYPTO_MODE_HMAC_MD5	MD5 HMAC mode
CRYPTO_MODE_HASH_SHA1	SHA-1 hash mode
CRYPTO_MODE_HMAC_SHA1	SHA-1 HMAC mode
CRYPTO_MODE_HASH_SHA224	SHA-224 hash mode (not to be confused with SHA-512_224)
CRYPTO_MODE_HMAC_SHA224	SHA-224 HMAC mode
CRYPTO_MODE_HASH_SHA256	SHA-256 hash mode
CRYPTO_MODE_HMAC_SHA256	SHA-256 HMAC mode
CRYPTO_MODE_HASH_SHA384	SHA-384 hash mode
CRYPTO_MODE_HMAC_SHA384	SHA-384 HMAC mode
CRYPTO_MODE_HASH_SHA512	SHA-512 hash mode
CRYPTO_MODE_HMAC_SHA512	SHA-512 HMAC mode
CRYPTO_MODE_HASH_SHA512_224	SHA-512_224 hash mode
CRYPTO_MODE_HMAC_SHA512_224	SHA-512_224 HMAC mode
CRYPTO_MODE_HASH_SHA512_256	SHA-512_256 hash mode
CRYPTO_MODE_HMAC_SHA512_256	SHA-512_256 HMAC mode
CRYPTO_MODE_HASH_SHA3_224	SHA3-224 hash mode
CRYPTO_MODE_HASH_SHA3_256	SHA3-224 hash mode
CRYPTO_MODE_HASH_SHA3_384	SHA3-224 hash mode
CRYPTO_MODE_HASH_SHA3_512	SHA3-224 hash mode
CRYPTO_MODE_HASH_SHAKE128	SHAKE128 hash mode
CRYPTO_MODE_HASH_SHAKE256	SHAKE256 hash mode
CRYPTO_MODE_HASH_CSHAKE128	CSHAKE128 hash mode

Hash Enumeration	Mode Description
CRYPTO_MODE_HASH_CSHAKE256	CSHAKE256 hash mode
CRYPTO_MODE_MAC_POLY1305	POLY1305 MAC mode
CRYPTO_MODE_MAC_XCBC	AES XCBC MAC mode
CRYPTO_MODE_MAC_CMAC	AES CMAC MAC mode
CRYPTO_MODE_MAC_KASUMI_F9	KASUMI F9 MAC mode
CRYPTO_MODE_MAC_KMAC128	KMAC128 MAC mode
CRYPTO_MODE_MAC_KMAC256	KMAC256 MAC mode
CRYPTO_MODE_MAC_KMACXOF128	KMACXOF128 MAC mode
CRYPTO_MODE_MAC_KMACXOF256	KMACXOF256 MAC mode
CRYPTO_MODE_MAC_SNOW3G_UIA2	SNOW3G UIA2 MAC mode
CRYPTO_MODE_MAC_ZUC_UIA3	ZUC UIA3 MAC mode
CRYPTO_MODE_SSLMAC_MD5	SSL 3.0 MAC based on MD5
CRYPTO_MODE_SSLMAC_SHA1	SSL 3.0 MAC based on SHA1
CRYPTO_MODE_HASH_CRC32	CRC32 HASH (not cryptographically secure)
CRYPTO_MODE_MAC_MICHAEL	Michael-MIC Algorithm

3.4.2 spacc_write_context()

Once a SPAcc job index has been acquired that maps to a context on a device, the keys (and other salient pieces of data such as IV and salt) can be loaded with the *spacc_write_context()* function.

```
int spacc_write_context (spacc_device *spacc, int handle, int op,
unsigned char * key, int ksz, unsigned char * iv, int ivsz);
```

The function assigns values to the context identified by *job_idx* and operates on the cipher (and RC4) or hash context pages depending on the value of *op*. When set to **SPACC_CRYPTO_OPERATION** it sets the cipher/RC4 pages, when set to **SPACC_HASH_OPERATION** it sets the hash pages.

The key is passed as *key* of *ksz* bytes length, and the IV/salt as *iv* of *ivsz* bytes length. In certain modes, the lengths are implicit. See the *DWC SPAcc Databook,* for more details.

When calling this function to load a fully scheduled RC4 key, the key (cipher mode **CRYPTO_MODE_RC4_KS**) is assumed to be 258 bytes long. The first two bytes are *i* and *j* respectfully (see *spacc_write_rc4_context*()) and the remaining 256 bytes are the shuffle array. The calling application is responsible for scheduling the key as the SDK does not contain any cryptographic software.

In typical combined modes, the caller must call this function twice to initialize the two different context pages. In some cases it may be useful to read back the context data. The function *spacc_read_context()* should be used.

```
int spacc_read_context (spacc_device *spacc, int job_idx, int op,
unsigned char *key, int ksz, unsigned char *iv, int ivsz);
```

Reading back the context allows for device context swapping. The typical recycling of a context would be:

1. Call *spacc_open*().
2. Call *spacc_write_context*() once or twice depending on the job.
3. Call *spacc_set_operation*().
4. Perform SPAcc jobs.
5. Call *spacc_read_context*().
6. Call *spacc_close*().
7. Start a new job doing steps 1-6.
8. When you want to resume the first context simply write back the saved keys/ivs.

3.4.3 spacc_load_skp()

To use the secure key port functionality of the SPAcc, the key must be loaded with the *spacc_load_skp()* function.

```
int spacc_load_skp(spacc_device *spacc, uint32_t *key, int keysz, int idx, int alg,
                   int mode, int size, int enc, int dec);
```

This loads a key of *keysz* 32-bit words pointed to by *key* into the secure context indexed by *idx*. The *alg* and *mode* parameters are one of the mode enumerations for the SPAcc hardware listed in the *DWC SPAcc Databook* under the **CTRL** register. The *size* register is either 0, 1, or 2 for 64/128, 192, and 256-bit keys, respectively. The *enc* and *dec* registers indicate whether the context can be used for encryption and decryption respectfully.

3.4.4 spacc_set_operation()

Once a SPAcc context has been allocated the operating mode can be assigned with the *spacc_set_operation()* function.

```
int spacc_set_operation (spacc_device *spacc, int job_idx, int op, uint32_t prot,
uint32_t icvcmd,
                         uint32_t icvoff, uint32_t icvsz, uint32_t sec_key);
```

This function sets the encryption direction, ICV protocol, position, offset, and size. The *op* parameter should be set to **OP_ENCRYPT** for encryption or **OP_DECRYPT** for decryption.

The *prot* variable is set to one of the following three values: **ICV_HASH**, **ICV_HASH_ENCRYPT**, or **ICV_ENCRYPT_HASH**, which either produce a hash (or MAC tag) on the plaintext, and then encrypt the tag or on the ciphertext, respectively.

The *icvcmd* parameter must be set to one of: **IP_ICV_OFFSET**, **IP_ICV_APPEND**, or **IP_ICV_IGNORE**. These tell the engine to store the ICV (hash or MAC tag) at a fixed location, append to the ciphertext, or ignore the setting (for jobs that do not have a hash component).

The ICV offset is set by the *icvoff* parameter and only used in **IP_ICV_OFFSET** mode. The size of the ICV is passed as *icvsz* and can be used to tell the engine to truncate the (or expect a truncated) ICV.

The *sec_key* parameter is used to indicate whether secure key port is to be used.

3.4.5 spacc_write_rc4_context()

The RC4 context can be initialized by either loading a short 40- or 128-bit key in the key context page, or by loading a fully scheduled RC4 context (258 bytes long) into the RC4 context page. The fully scheduled key can be loaded with *spacc_write_context*() or with the following function directly. When calling *spacc_write_context*(), the key is assumed to be 258 bytes long, where the first two bytes are *i* and *j* and the remaining 256 bytes are *ctxdata*.

```
int spacc_write_rc4_context (spacc_device *spacc, int job_idx, unsigned char i,
unsigned char j,
unsigned char *ctxdata);
```

This sets the RC4 context page with the 256 bytes of state from *ctxdata* (the permutation) and the i and *j* pointers. This should only be called after opening a job_idx with the **CRYPTO_MODE_RC4_KS** mode, otherwise the engine will be programmed to expand a key context page key into an RC4 context page.

The RC4 context can be retrieved with the following function:

```
int spacc_read_rc4_context (spacc_device *spacc, int job_idx, unsigned char *i,
unsigned char *j,
                            unsigned char *ctxdata);
```

which retrieves the RC4 context page and stores the 256-byte permutation in *ctxdata* and the pointers in *i* and *j*.

3.4.6 spacc_set_key_exp()

This function sets the **KEY_EXP** bit of the SPAcc CTRL register. It is used in the following circumstances:

- Using AES in decrypt mode.
- Using RC4 in 40 or 128-bit key mode (but not **KS** mode).

```
int spacc_set_key_exp(spacc_device *spacc, int job_idx);
```

The function modifies the internal CTRL register of the job associated with *job_idx* to have the bit. By default, new jobs do **not** have this bit set.

3.4.7 spacc_set_auxinfo()

The *spacc_set_auxinfo()* function is used to set the AUXINFO register of the SPAcc (used by 3GPP modes).

```
int spacc_set_auxinfo(spacc_device *spacc, int job_idx, uint32_t direction, uint32_t
bitsize);
```

This sets the direction bit and bits used in the last byte in the AUXINFO register.

3.4.8 spacc_packet_enqueue_ddt()

The *spacc_packet_enqueue_ddt* function is the only function in the SDK that writes directly to SPAcc FIFO registers (through memory mapped IO). The caller must call the various support functions such as, *spacc_open()*, *spacc_set_context()*, *spacc_set_operation()*, and *spacc_set_auxinfo()* before calling this function.

```
int spacc_packet_enqueue_ddt (spacc_device *spacc, int job_idx,
                              pdu_ddt *src_ddt, pdu_ddt *dst_ddt,
                              uint32_t proc_sz,
                              uint32_t aad_offset, uint32_t pre_aad_sz, uint32_t
post_aad_sz,
                              uint32_t iv_offset, uint32_t prio);
```

This function requires previously initialized DDT tables (using PDU DDT functions provided by the platform device driver). Passed as *src_ddt* and *dst_ddt*. The only thing that this function does with them is read out the *len* and *phys* members to program SPAcc FIFO registers.

The *proc_sz* parameter indicates the desired value for the PROCLEN register. It can be set to the special value **SPACC_AUTO_SIZE** to allow the function to compute the register on the fly. Setting it to a fixed value is useful, when processing partial blocks of a larger mapped job.

The *aad_offset* parameter indicates the offsets for the source and destination buffers to the core.

Note	It must be pre-formatted as per the *SPAcc Databook*. Usually, this value is left as zero.

The *pre_aad_sz* and *post_aad_sz* parameters indicate the size of the pre- and post-aad, respectively. By default, the SDK does not copy AAD data to the destination buffer. This can be changed by or'ing the value **SPACC_AADCOPY_FLAG** into the *pre_aad_sz* parameter. Without the flag, setting the *pre_aad_sz* is most commonly used by hashing jobs by setting it to *proc_sz* such that only the hash output is emitted to the destination buffer.

The *iv_offset* parameter indicates the offset of the IV (if to be read from the source DDT mapped data) or zero if to be read from the context page. The MSB (0x80000000) must be set if IV import is to be used (allowing for an IV offset of zero).

The *prio* parameter indicates, which priority queue to fire the job in (if supported).

Once a job has been pushed on the command FIFO it can either be polled to be dequeued or the user can wait for an interrupt (depending on the platform setup). The user may not alter the contents of their DDT structures until the job is complete.

The function will return **CRYPTO_FIFO_FULL,** if the CMD FIFO is full.

The function will return **CRYPTO_CMD_FIFO_INACTIVE,** if the priority specified refers to a CMD FIFO that is not present.

3.4.9 spacc_packet_dequeue()

The *spacc_packet_dequeue()* function polls the SPAcc to determine if the specified job is complete. In reality, the function sees, if *any* job is ready to be popped off the status FIFO and it marks that respective job as complete, upon returning it indicates to the caller if the job they inquired about is complete.

```
int spacc_packet_dequeue (spacc_device *spacc, int job_idx);
```

This function returns **CRYPTO_OK,** if the job specified by *job_idx* is complete or **CRYPTO_INPROGRESS** if it is still pending.

3.4.10 spacc_pop_packets()

This function checks the status FIFO and pops all available jobs off. It marks the job complete and returns the number of jobs popped. The address to and integer must be passed in. The return code is **CRYPTO_OK** or **CRYPTO_INPROGRESS**.

```
int spacc_pop_packets (spacc_device * spacc, int *num_popped);
```

3.4.11 spacc_close()

Once a job has completed and a context is no longer required, it should be freed as soon as possible to return the context to the device for other jobs/users use.

```
int spacc_close (spacc_device *spacc, int job_idx);
```

This closes the context and returns it to the available pool. Note that this will not wait for the job to pop off the status FIFO.

3.5 IRQ Functions

The following functions abstract the details and management of the IRQ registers.

3.5.1 spacc_irq_XXX_enable/disable()

The following functions abstract the details of managing the IRQ registers. Check the HW documentation, to understand when the IRQ signals are triggered.

The *cmdx* parameter represents the CMD0, CMD1, or CMD2 register in the hardware. The number of CMD options depends on enabling QOS in hardware. The *cmdx_cnt* is a threshold level for when to trigger the interrupt.

```
void spacc_irq_cmdx_enable (spacc_device *spacc, int cmdx, int cmdx_cnt);
void spacc_irq_cmdx_disable (spacc_device *spacc, int cmdx);
```

These function sets the threshold for triggering the IRQ at the same time as enabling the interrupt.

```
void spacc_irq_stat_enable (spacc_device *spacc, int stat_cnt);
void spacc_irq_stat_disable (spacc_device *spacc);
```

These functions enable and disable the STAT_WD IRQ.

```
void spacc_irq_stat_wd_enable (spacc_device *spacc);
void spacc_irq_stat_wd_disable (spacc_device *spacc);
```

These functions enable and disable the RC4 DMA IRQ.

```
void spacc_irq_rc4_dma_enable (spacc_device *spacc);
void spacc_irq_rc4_dma_disable (spacc_device *spacc);
```

These functions enable and disable the Global setting.

```
void spacc_irq_glbl_enable (spacc_device *spacc);
void spacc_irq_glbl_disable (spacc_device *spacc);
```

3.5.2 spacc_process_irq()

This function determines, which IRQ was signaled and calls the registered callback function. The callback functions must be initialized in the spacc_device structure that is passed to the function. These parameters are:

```
void (*irq_cb_cmdx)(struct _spacc_device *spacc, int x);
void (*irq_cb_stat)(struct _spacc_device *spacc);
void (*irq_cb_stat_wd)(struct _spacc_device *spacc);
void (*irq_cb_rc4_dma)(struct _spacc_device *spacc);
```

and the function definition is:

```
uint32_t spacc_process_irq(spacc_device *spacc);
```

3.6 Other Functions

3.6.1 spacc_error_msg()

The SPAcc SDK returns CRYPTO_OK from most functions if successful, or a negative number to indicate failure. The error codes can be mapped to printable strings with the following function:

```
unsigned char *spacc_error_msg (int err);
```

3.6.2 spacc_set_secure_mode()

The *spacc_set_secure_mode()* function sets the secure mode register of the SPAcc, which allows for restrictions on how the bus mastering and slave register map access behave.

```
void spacc_set_secure_mode (spacc_device *spacc, int src, int dst, int ddt, int
global_lock);
```

The parameters all behave as Booleans, where *src* controls whether source data mastering asserts secure mode, *dst* for destination data, *ddt* for DDT structure access, and *global_lock* for the slave port registers.

3.6.3 spacc_dump_ctx()

The spacc_dump_ctx() function is used to get debugging information about a given SPAcc context.

```
void spacc_dump_ctx(spacc_device *spacc, int ctx);
```

This will print through ELPHW_PRINT the contents of the hash, cipher, and if appropriate the RC4 context for a given context specified. If the context is not accessible (secured context from normal port) it will print out all zeroes.

3.6.4 spacc_isenabled()

This function is provided to determine, if a given cipher or hash mode is enabled in the core.

```
int spacc_isenabled(spacc_device *spacc, int mode, int keysize);
```

The user passes the CRYPTO_MODE_* mode they wish to test along with the keysize in bytes passed as *keysize*. It will return 0, if the mode is not enabled or non-zero if it is.

3.6.5 spacc_compute_xcbc_key()

This function is provided to initialize a 3-key XCBC MAC key from a single 16-byte key. This function is typically used in protocols such as, IPsec where the SA supplied key is short. This function requires that either AES-128-ECB or AES-128-CBC be enabled to correctly compute the 48-byte key.

```
int spacc_compute_xcbc_key(
           spacc_device *spacc,
           int           job_idx,
    const unsigned char *key, int keylen,
          unsigned char *xcbc_out);
```

This will use the n-byte input key pointed to by *key* of length *keylen* and process it into the 48-byte scheduled key *xcbc_out*.

The value of *job_idx* can be a SPAcc handle that you intend to associate with the programmed key. This will remove the need to allocate a second context simply to compute the working key.

Note that this will overwrite the cipher key for that SPAcc handle's context. Therefore, it is important to call this before programming the working ciphering key to the SPAcc context page.

3.7 SPAcc Userspace Interface

On top of the *src/core/kernel/example.c* example routines is a userspace interface that allows cipher, hash, and combined cipher/hash jobs to be performed from a userspace task. The module source code is *src/core/kernel/spacc_dev.c* and builds as *elpspaccusr.ko*. When loaded the driver provides the entry */dev/spaccusr*, which the user space API can then use to perform SPAcc jobs.

The userspace side of the API resides in *src/core/user/spaccdev.c* and can be linked into any application that wishes to make use of the SPAcc. This source file depends on two additional files found in *src/core/include*: *elpspaccusr.h* and *elpspaccmodes.h*, which can be copied with *spaccdev.c* into a user project.

Packets processed by the driver occur in a blocked mode in which the function will only return when the job is complete (or an error detected). It will re-try jobs internally, if the command FIFO is full allowing more jobs to be on the fly than FIFO entries.

Note	By default, the userspace driver allows up to 32 segments per source and destination buffer (including user side scatter gather). This typically is more than enough to handle up to 64 Kbyte jobs using 4 K pages. The user can change this by defining the variable **SPACC_DEFAULT_DDT_LENGTH** in their environment or editing the definition at the top of *synopsys/driver/src/core/kernel/spacc_dev.c*.

3.7.1 Opening a Userspace Handle

```
int spacc_dev_open(
   struct elp_spacc_usr *io,
   int cipher_mode, int hash_mode,    // cipher and hash (see elpspaccmodes.h)
   int encrypt,                       // non-zero for encrypt (sign) modes
   int icvmode,                       // ICV mode (see elpspaccmodes.h)
   int icvlen,                        // length of ICV, 0 for default length (algorithm
dependent)
   int aad_copy,                      // non-zero to copy AAD to dst buffer
   unsigned char *ckey, int ckeylen, unsigned char *civ, int civlen,  // cipher key and
IVs
   unsigned char *hkey, int hkeylen, unsigned char *hiv, int hivlen); // hash key and
IVs (if any)
```

This function opens a SPAcc device handle. The cipher/hash modes are specified the same as if calling *spacc_open()* in the kernel space. The *encrypt* flag is set to non-zero, when encrypting (generating a hash/hmac output) and zero when decrypting (verifying a hash/hmac input). The *icvmode* is set to one of the **ICV_** flags defined in *elpspaccmodes.h*. The desired length of the ICV may be specified as *icvlen* (bytes), where 0 indicates to use the default (maximum) length. The user may wish to indicate that AAD is to be copied through the *aad_copy* flag.

The ciphering key and IV are passed as *ckey* and *civ* with their respective lengths in bytes *ckeylen* and *civlen*.

The hashing key and IV are passed as *hkey* and *hiv*, respectively (same with lengths).

When certain AES modes are used in decrypt mode, the open call will cause a short SPAcc job to occur to ensure the cipher key has been properly initialized. This is done to allow multiple handles to be bound to this handle correctly. This should be transparent for the user but they will notice an extra IRQ during the open call.

The function returns 0 upon success and a negative number upon error.

3.7.2 Performing a SPAcc Job

A cipher, hash, or cipher/hash job may be performed with this API with the following function:

```
int spacc_dev_process(
   struct elp_spacc_usr *io,
   unsigned char *new_iv,
   int            iv_offset,
   int            pre_aad,
   int            post_aad,
   int            src_offset,
   int            dst_offset,
   unsigned char *src, unsigned long src_len,
   unsigned char *dst, unsigned long dst_len);
```

This function instructs the device driver to process a SPAcc job and does not return (it is a blocking call) until the SPAcc completes the job or there is an error.

The user may use a fresh cipher IV with the job by passing a pointer to the new IV in *new_iv*. If they wish to use IV import or the existing IV in the cipher key context, they may pass it as **NULL**.

The user may instruct the device to use the IV import functionality to read an IV from the source packet by setting the *iv_offset* parameter. If not they may set it to the default of *-1*, which means to ignore the IV import functionality.

Note	It is more efficient to use the IV import functionality over passing an IV pointer since it invokes fewer slave port accesses to the SPAcc core. If at all possible, it is better to store the IV at the beginning or end of the source buffer (if there is room) and import it from there.

The length of the pre- and post-AAD data is specified in the *pre_aad* and *post_aad* parameters.

The offset from the start of the source and destination buffers to the first byte of *pre_aad* (or plaintext/ciphertext if *pre_aad* is zero) is specified in the *src_offset* and *dst_offset* parameters.

The length of the input packet is specified in *src_len*. It is much like the **PROC_LEN** register of the SPAcc in that it includes pre-AAD but not post-AAD. Unlike the **PROC_LEN** register it never includes the ICV data (for example, when *icv_mode* is **ICV_HASH_ENCRYPT**). The *src_len* does not have to contain the IV, if IV import is being used. For instance, if the IV is stored, passed the end of the packet data, the driver will know to map ***more*** than the specified bytes.

The *dst_len* parameter is how many bytes starting from *dst* to map into the kernel.

Note	It is more efficient to perform SPAcc jobs in-place, where *src* == *dst*. In this mode the Kernel will only map the larger of the input/output buffers once from userspace into the kernel. Users can still use source/destination offsets but the destination offset cannot overlap with the source buffer (unless the offsets are equal).

If the function is successful it will return the number of bytes emitted by the SPAcc (which may be zero, if there are no plaintext bytes and *aad_copy* is set to zero). It will return -1 upon error. The user may check the variable *io->io.err* for the SPAcc related error.

Note	The kernel side of this function will retry up to 100 times with a 10 millisecond delay[1] to program the job if the CMD FIFO is full. After the 100th retry it will return an error with *io->io.err* set to **CRYPTO_FIFO_FULL**.

See *src/core/user/spaccdevtest.c*, for an example of calling this API in one of the three modes.

[1] It is ideal to have the clock tick set to at least 100 Hz for this function to have a ceiling of 1 sec timeout.

3.7.3 3GPP Ciphersuite Modes

With the 3G cipher suites (SNOW3G, ZUC, KASUMI F9) users will need to call, the following macro to set the AUXINFO register.

```
spacc_dev_setaux(fd, align, dir),
```

which assigns the alignment and direction flags to the state pointed to by *fd*.

3.7.4 Performing Jobs with Userspace Scattergather

The function *spacc_dev_process()* only allows the user to pass a single buffer for source and/or destination to the kernel side of the task. This is sufficient for many APIs but prevents users from gathering jobs from distinct sources. To support scattergather users from the userspace there is access to create DDT entries and pass them on to the kernel.

The following two macros are used to set DDT entries:

```
ELP_SPACC_SRC_DDT(fd, x, addr, len)
ELP_SPACC_DST_DDT(fd, x, addr, len),
```

where *fd* is a *struct elp_spacc_ioctl* type, *x* is which entry to set and *addr*/*len* are the address/length of this segment. DDT lists must be terminated with a NULL pointer, which can be accomplished with:

```
ELP_SPACC_SRC_TERM(fd, x)
ELP_SPACC_DST_TERM(fd, x),
```

which sets the *x*'th entry to the terminator. The macro **ELP_SPACC_USR_MAX_DDT** indicates the maximum number of DDT entries and can be changed (provided the kernel module is recompiled as well). By default, it is set to 8.

Note that room is provided for the required terminator, so by default nine DDT entries are allocated.

Once the DDTs are ready, the source and destination lengths must be computed. These are the original *src_len*/*dst_len* passed to *spacc_dev_process()*. They can be set manually with:

```
ELP_SPACC_SRCLEN_SET(fd, x)
ELP_SPACC_DSTLEN_SET(fd, x)
```

or set to be computed on the fly with:

```
ELP_SPACC_SRCLEN_RESET(fd)
ELP_SPACC_DSTLEN_RESET(fd)
```

Note that if they are computed on the fly, the caller must reset them every time they issue a job.

Unlike with *spacc_dev_process()*, the caller is required to include the IV (if using *iv_offset*) in one of the DDT segments passed. Users can still perform jobs in place (for higher performance) provided both the destination and source DDTs are equal (in pointers and in lengths).

Once the DDTs and source and destination lengths are ready, the user can call the following function to issue the job:

```
int spacc_dev_process_multi(
   struct elp_spacc_usr *io,
   unsigned char *new_iv,
   int      iv_offset,
   int      pre_aad,
   int      post_aad,
   int      src_offset,
   int      dst_offset
   int      map_hint);
```

This will issue the job with the same semantics as *spacc_dev_process()*. The new parameter *map_hint* is used to help the kernel module determine, if the mappings overlap. It may be one of the following values: **SPACC_MAP_HINT_TEST, SPACC_MAP_HINT_USESRC, SPACC_MAP_HINT_USEDST, SPACC_MAP_HINT_NOLAP**.

The **SPACC_MAP_HINT_TEST** hint instructs the module to test, whether the mappings are compatible with a single mapping. The **SPACC_MAP_HINT_USESRC** and **SPACC_MAP_HINT_USEDST** instructs the module to use the source or destination mappings explicitly. The caller must use the larger of the two mappings, otherwise, the job could fail (and the application could segfault). The **SPACC_MAP_HINT_NOLAP** hint instructs the module that there is no overlap and it must map them separately.

For the general case use **SPACC_MAP_HINT_TEST** as that will safely test internally, if the mapping is compatible (though it will be slower than an explicit hint).

Note	Calls to *spacc_dev_process()* and *spacc_dev_process_multi()* may be mixed on the same handle provided the DDT entries are initialized prior to each call to *spacc_dev_process_multi()*. Calling *spacc_dev_process()* **will overwrite** the DDT entries associated with the handle.

An example of this API can be found in the function *speed()* of the file *synopsys/driver/src/core/user/spaccdevtest.c*.

3.7.5 Context Sharing

It is possible to share SPAcc key context pages between jobs to allow parallel use of the SPAcc (through threading on the user side) through the register and bind API. First, the caller must register their state for sharing by calling:

```
int spacc_dev_register(struct elp_spacc_usr *io);
```

This assigns the state a random 128-bit key[2] that is later used to pair open handles. This function returns 0 upon success. Each time it is called a new 128-bit key is used for sharing.

Note that this key has nothing to do with whatever ciphering key is being used, it is only used to pair handles. Essentially, this key prevents other threads/processes from attempting to bind to an open handle and then using the key context without permission.

Next, a new handle can be opened and paired (bound) to a master by calling:

```
int spacc_dev_open_bind(struct elp_spacc_usr *master, struct elp_spacc_usr *new);
```

This will clone *master* and initialize *new*. When done both refer to the same SPAcc key context page, despite using different SPAcc handles. Now both *master* and *new* can be used in parallel (in different threads) to allow filling the CMD FIFO of the SPAcc.

The SPAcc key context page is not freed until all references are released so the *master* handle can be closed while the *new* one is still operational.

Note	With shared contexts the IV import functionality must be used with every packet since the key context page cannot be changed. This is accomplished by passing *iv_offset* as a value above -1 to the function *spacc_dev_process()*. The use of the *new_iv* parameter is **not** allowed in this mode.

The program in *synopsys/driver/src/core/user/spaccdevtest.c* makes use of this functionality towards the end of the file in the function *speed()*. Another way to fit this into APIs is to use an offset, where the application expects to see application data, for instance:

```
struct data_pointer {
    unsigned char *mem, // base where we would put the IV
                  *src; // where application data goes
};
```

which can then be initialized as follows:

```
int init_dp(struct data_pointer *dp, int len)
{
   dp->mem = calloc(1, len + 16);
   dp->src = dp->mem + 16;
   return dp->mem == NULL;
}
```

Now the application can read/write data from *dp->src* and the provider can pass *dp->mem* to the SPAcc userspace API. Here, leave 16 bytes, so IV can be stored and use the IV import functionality easily.

[2] By reading /dev/urandom for 16 bytes.

3.7.6 Partial Message Processing

For SPAcc cores that support partial message processing users may program jobs through the userspace API in partial steps. This allows large jobs to be processed through the SPAcc in steps that are more manageable. Jobs are programmed as normal with the exception of message flags that must be used.

```
#define ELP_SPACC_USR_MSG_DEFAULT(fd)
#define ELP_SPACC_USR_MSG_FIRST(fd)
#define ELP_SPACC_USR_MSG_MIDDLE(fd)
#define ELP_SPACC_USR_MSG_LAST(fd)
```

The first block of data processed must be processed by calling **ELP_SPACC_USR_MSG_FIRST()** on the *elp_spacc_usr* handle that is open. If there are more than one block remaining, **ELP_SPACC_USR_MSG_MIDDLE()** must be called (only once), and for the final block **ELP_SPACC_USR_MSG_END()**. For the rules of how to split up jobs for partial processing, refer to the ***DWC SPAcc Databook***.

To return an open handle to the default non-partial processing mode use **ELP_SPACC_USR_MSG_DEFAULT()**.

For a simple example of partial processing, refer to *synopsys/driver/src/core/user/spaccdevtest.c*.

3.7.7 Feature Testing

The device driver may be queried to determine, which cipher and hash modes are available to the user.

```
int spacc_dev_features(struct elp_spacc_features *features);
```

This will populate the structure *elp_spacc_features* with values from the kernel device driver. It will return non-zero upon error.

The structure has the following definition:

```
struct elp_spacc_features {
   unsigned
       project,
       partial,
       qos,
       max_msg_size;
   unsigned char
       modes[CRYPTO_MODE_LAST];
};
```

The *project* value denotes the SPAcc project number used to identify the configuration. The *partial* value denotes whether partial processing is supported or not. The *qos* flag indicates whether IV import functionality is available. The *max_msg_size* value denotes the maximum message size that is allowed by the core.

To determine if certain cipher or hash modes are available, the following function may be used:

```
int spacc_dev_isenabled(struct elp_spacc_features *features, int mode, int keysize);
```

This function returns 1 if the mode is available. The mode is specified as one of the **CRYPTO_MODE_*** values defined in *synopsys/driver/src/core/include/elpspaccmodes.h* and the key size in bytes.

For example, to determine if AES-128-CBC is available:

```
if (spacc_dec_isenabled(&features, CRYPTO_MODE_AES_CBC, 16)) {
   printf(“AES-128-CBC is available\n”);
}
```

When testing the presence of unkeyed HASH functions, use the key size 0. For example,

```
if (spacc_dec_isenabled(&features, CRYPTO_MODE_HASH_MD5, 0)) {
   printf(“MD5 hash is available\n”);
}
```

would test the presence of the MD5 hash function.

3.7.8 Closing Handles

```
int spacc_dev_close(struct elp_spacc_usr *io);
```

This function closes the device handle.

4 DWC PKA

4.1 Initialization

The PKA SDK provides two main header files:

```
#include “elppka_hw.h” /* Constants for the hardware register offsets & fields */
#include “elppka.h” /* Declarations for this SDK functionality */
```

Most PKA functions take a pointer to the device state structure as their first argument. Each state structure tracks a particular hardware instantiation. Before any other SDK functions to access the hardware can be used, the structure must first be initialized by calling

```
int elppka_setup(struct pka_state *pka, uint32_t *regbase);
```

where

- `pka` points to the allocated state structure,
- `regbase` is the base address of the PKA's register map.

This function will put the device into a sane initial state and fill out the state structure (described below). The function returns 0 on success. Defined errors are:

- `CRYPTO_NOT_IMPLEMENTED`, if the SDK was not able to parse the hardware config registers.

4.1.1 Device Configuration

Some of the PKA's features may be affected by the hardware configuration. When the device is initialized by `elppka_setup`, the hardware config registers are parsed to fill out fields in the state structure, which can be inspected to influence software behavior. These parameters are accessible through the `cfg` member of the state structure:

```
struct pka_state {
   struct pka_config {
      unsigned alu_size, rsa_size, ecc_size;h
      unsigned fw_ram_size, fw_rom_size;
      unsigned ram_offset, rom_offset;
   } cfg;

   /* ... */
};
```

The meaning of the `pka_config` struct members is:

- `alu_size`: Configured width of the PKA's ALU. Does not affect device functionality, but can be used for informational purposes.
- `rsa_size`: Maximum operand width of RSA functionality, in bits.
- `ecc_size`: Maximum operand width of ECC functionality, in bits.
- `fw_ram_size`: Size of the firmware RAM, in 32-bit words.
- `fw_rom_size`: Size of the firmware ROM, in 32-bit words.
- `ram_offset:` Offset of the firmware RAM in the memory map, as a count of 32-bit words.
- `rom_offset`: Offset of the firmware ROM in the memory map, as a count of 32-bit words.

Either fw_ram_size or fw_rom_size may be zero, which indicates that there is no firmware memory of the corresponding type. If *both* members are zero, this indicates that the PKA has been configured to use a custom external firmware memory, which is out of scope of this SDK.

4.2 Firmware Handling

The PKA SDK provides functionality for loading firmware images into a device and to help the caller find entry points at runtime. Use of this functionality is optional. These functions load from a binary image format, which is a subset of the 32-bit ELF format, and images can be processed with ELF tools commonly available on modern operating sytstems.

4.2.1 Firmware Structure

The functions in this section use a structure to store the firmware details. The application is responsible for allocating storage for the firmware structure and passing it to all firmware functions.

```
struct pka_fw {
   unsigned long ram_size, rom_size;
   struct pka_fw_tag ram_tag, rom_tag;

   const char *errmsg;
};
```

where

- `ram_size` specifies the size of the image's RAM section, in 32-bit words. This member is 0, if the image does not have a RAM section.
- `rom_size` specifies the size of the images ROM section, in 32-bit words. This only includes the tag block, and does not include any actual instructions. This member is 0, if the image does not have a ROM section.
- `ram_tag` contains the broken-out fields of the RAM tag, for informational purposes. Only valid, if ram_size is non-zero.
- `rom_tag` contains the broken-out fields of the ROM tag, for informational purposes. Only valid, if rom_size is non-zero.
- `errmsg` points to a human-readable error message describing the last error. This member is only defined, when the most recent function result was `CRYPTO_INVALID_FIRMWARE`.

4.2.2 Reading a Firmware Image

To parse an image, simply load it into a buffer and call `elppka_fw_parse`. This function will fill in the firmware structure and copy the relevant sections out of the image data.

```
int elppka_fw_parse(struct pka_fw *fw, const unsigned char *data, unsigned long len);
```

where

- `fw` points to an allocated firmware structure,
- `data` points to the buffer containing the image data,
- `len` specifies the length of the image, in bytes.

Returns 0 on success. Defined errors are:

- `CRYPTO_INVALID_FIRMWARE,` if the firmware image is invalid. The *errmsg* member of the firmware struct will be updated with a human-readable description of the error.

The original buffer is not used after calling `elppka_fw_parse`, so it may be safely freed.

When you are finished with the PKA firmware, it should be cleaned up by calling the function

```
void elppka_fw_free(struct pka_fw *fw);
```

where

- `fw` points to an initialized firmware structure.

This function will free any internal buffers allocated by `elppka_fw_parse`.

4.2.3 Loading a Firmware Image

After successfully parsing an image, it can be loaded into the hardware. For firmware images with a RAM component, this will write the instructions into the engine's firmware RAM. For images with a ROM component, this will validate the ROM's tag block against the tag in the image to ensure that they match. This loading is performed by calling the function

```
int elppka_fw_load(struct pka_state *pka, struct pka_fw *fw);
```

where

- `pka` points to an initialized PKA state structure,
- `fw` points to an initialized firmware structure.

Returns 0 on success. Defined errors are

- `CRYPTO_INVALID_FIRMWARE,` if the firmware cannot be loaded onto the device, either because there is not enough RAM to fit the image or the ROM section did not validate successfully. The *errmsg* member of the firmware structure will be updated with a human-readable description of the error.

4.2.4 Entry Point Lookup

Once an image is parsed, it can be used to lookup entry points by name. Using this functionality allows you to write applications that can be more easily adapted to different firmwares. To lookup an entry point, call the function

```
int elppka_fw_lookup_entry(struct pka_fw *fw, const char *entry);
```

where

- `fw` points to an initialized firmware structure,
- `entry` designates the name of the entry point.

Returns the non-negative entry point on success, which can be used as the entry argument to elppka_start. Defined errors are:

- `CRYPTO_INVALID_FIRMWARE`, if the firmware image is invalid. The errmsg member of the firmware struct will be updated with a human-readable description of the error.
- `CRYPTO_NOT_FOUND`, if the specified entry point was not found in the image.

The entry point names are specified in the firmware documentation; examples are "`modexp`" and "`pmult`".

4.3 Operands

When performing an operation on the PKA, one must first load the operand data into the engine. The PKA SDK provides two helper functions to do this: elppka_load_operand reads data from a buffer and loads it into the engine, and elppka_unload_operand reads data from the engine and writes it out to a buffer. The functions are:

```
int elppka_load_operand(struct pka_state *pka, unsigned bank, unsigned index,
                                               unsigned size, const uint8_t *data);
int elppka_unload_operand(struct pka_state *pka, unsigned bank, unsigned index,
                                                 unsigned size, uint8_t *data);
```

where

- `pka` points to an initialized state structure,
- `bank` is one of PKA_OPERAND_A, PKA_OPERAND_B, PKA_OPERAND_C, or PKA_OPERAND_D,
- `index` is the operand index within the specified bank,
- `size` is the operand size, in bytes,
- `data` points to the buffer used to load/unload operand data.

These functions return 0 on success. Defined errors are

- `CRYPTO_INVALID_SIZE` if the size parameter was invalid,
- `CRYPTO_INVALID_ARGUMENT` if the bank parameter was invalid,
- `CRYPTO_NOT_FOUND` if the index parameter was out of range for the specified device, bank and size combination.

The meaning of PKA operands depends on the firmware in use; see the documentation included with your firmware image for details.

4.3.1 PKA Endian

Internally, the PKA stores operands as a sequence of 32-bit words, with the least significant word first. On the other hand, software convention is usually for large integers to be stored as a sequence of bytes, with the most significant byte first. The PKA SDK uses that software convention for loading and unloading operands, and the data buffer is assumed to be stored with the most significant byte first. The SDK will convert between this format and the PKA internal format automatically. However, the hardware byte swapper must be configured properly by the host software prior to loading any operands. Normally this is configured automatically by `elppka_setup`, but it can be manually configured by calling the function

```
void elppka_set_byteswap(struct pka_state *pka, int swap);
```

where

- `pka` points to an initialized state structure,
- `swap` selects the mode: pass 0 to disable the byte swapper, or a non-zero value to enable it.

Little endian hosts should *enable* the byte swapper, whereas big endian hosts should *disable* it.

4.4 Running the PKA

To execute an operation on the PKA, first load any relevant operands into the engine, then call elppka_start to begin processing. This function is defined as

```
int elppka_start(struct pka_state *pka, uint32_t entry, uint32_t flags, unsigned size);
```

where

- `pka` points to an initialized state structure,
- `entry` specifies the firmware entry point (as an offset in 32-bit words),
- `flags` provides an initial setting for the PKA flags register, which will usually be set to zero,
- `size` specifies the operand size in bytes.

The list of firmware entry points is included with the firmware data, and the meaning of the various flags is documented in the *DWC Public Key Accelerator Firmware User Guide*. Most operations do not have any defined flags so the flags parameter should normally be zero, but some functions can be tweaked by setting flags to the bitwise-OR of one or more of the macros `PKA_FLAG_F1, PKA_FLAG_F2, PKA_FLAG_F3` or `PKA_FLAG_F4`. As a special case, if size is exactly 66, `PKA_FLAG_F1` will be automatically set to enable ECC-521 mode in firmware, which supports it.

The engine signals completion through an interrupt. See the *DWC Public Key Accelerator Databook* for details on accessing interrupt status. Once the interrupt is received and acknowledged, the result can be obtained by calling the function

```
int elppka_get_status(struct pka_state *pka, unsigned *code);
```

where

- `pka` points to an initialized state structure,
- `code` points to the location where the exit status will be written.

Returns 0 on success. Defined errors are

- `CRYPTO_INPROGRESS` if the engine is currently executing a job.

The meaning of the exit codes are defined in the PKA and firmware documentation, but generally an exit status of 0 indicates that the operation completed successfully and a non-zero value indicates failure.

4.4.1 Example Code

There are several reference programs in the *src/pka/user/* directory, which use the PKA driver. Note if the PKA firmware built by the corekit does not incorporate the operands required by the sample code, a failure will be reported.

- dumbtest.c: Demonstrates a modular addition.
- eccblind.c: Shows blinding of a point multiply.
- eccsignature.c: Demonstrates a simplified ECC sign and verify.
- memtest.c: Tests the PKA's memories.
- opdump.c: Dumps all of the PKA operands.
- poison.c: Wites non-sensical data to the PKA operands.
- simple521.c: Verifies a point is on the ECC curve.
- simpletest.c: Performs various PKA operations on an Edwards curve.

5 DWC TRNG

This section describes the API for the DWC TRNG. This is a new hardware design with an entirely new programming model.

5.1 Initialization

The DWC TRNG SDK provides two main header files:

```
#include “elpclp800_hw.h” /* Constants for the hardware register offsets & fields */
#include “elpclp800.h” /* Declarations for this SDK functionality */
```

All DWC TRNG functions take a pointer to the SDK state structure as their first argument. Each state structure tracks a particular hardware instantiation. Before any other SDK functions can be used, the structure must first be initialized by calling

```
void elpclp800_setup(struct elpclp800_state *clp800, uint32_t *regbase);
```

where

- clp800 points to the allocated state structure,
- regbase is the base address of the DWC TRNG's register map. This can be NULL, when using custom register I/O as described below.

This function will put the device into a sane initial state and fill out the state structure (described below).

5.2 Return Codes

Some of the DWC TRNG SDK functions are documented to indicate successful completion by their return value. All such functions have failure conditions indicated (as usual) by a negative return. If the function description documents have specific return codes, they are returned in the indicated situation. Nevertheless, a function may still fail for reasons not documented in its description by returning other error codes.

5.3 Hardware Parameters

The elpclp800_setup function will query the hardware to determine, what features are available in the design. The results are written to the state structure, and the application may inspect these values.

```
struct elpclp800_state {
     /* H/W features */
   unsigned short epn, stepping, output_len, diag_level;
   unsigned secure_reset:1, rings_avail:1;

/* ... */
};
```

The meaning of the struct members is:

- `epn`: Synopsys Project Number, an identifier associated with the particular design.
- `stepping`: Revision of this design configuration.
- `output_len`: Maximum supported output length of the core per random request, in bytes. In current designs, the core can be configured to support either 16 or 32-byte output.
- `diag_level`: Describes the amount of instrumentation in the configuration. Configurations with levels greater than zero allow some access to the internal state of the engine, when it is configured in secure mode, through the indirect access port.
- `secure_reset`: 1 if the design resets in secure mode, 0 otherwise.
- `rings_avail`: 1 if the design supports true random reseed mode (TRNG configuration), 0 in PRNG only configurations.

5.4 Register Access

Two functions are provided to access DWC TRNG registers directly:

```
uint32_t elpclp800_readreg(struct elpclp800_state *clp800, unsigned offset);
void elpclp800_writereg(struct elpclp800_state *clp800, unsigned offset, uint32_t val);
```

where

- `clp800` points to an initialized state structure,
- `offset` is the register index relative to the base address (in units of 32-bit words),
- (for `elpclp800_writereg`) val is the value to write.

For details on the available registers, see the *DWC True Random Number Generator Databook*.

5.4.1 Custom Register Wrappers

The above functions should be used for all register I/O as they allow the use of custom wrappers for access. This feature is useful when the DWC TRNG is instantiated in another design that does not expose its register map as a flat region of memory. For most configurations, it is not necessary to use the wrappers described in this section.

When `elpclp800_setup` is called with a `NULL` value for regbase, the custom wrappers in the state structure will be used. There are three struct members, which must be initialized manually prior to calling `elpclp800_setup`:

```
struct elpclp800_state {
   /* Register access */
   void (*writereg)(void *base, unsigned offset, uint32_t val);
   uint32_t (*readreg) (void *base, unsigned offset);
   void *regbase;

   /* ... */
}
```

All register accesses in the SDK are done indirectly through the writereg and readreg members in the state structure. The function assigned to the `readreg` member shall read the specified register and return the resulting value, the function assigned to the `writereg` member shall write the specified register with the given value. The `regbase` member in the state structure may be assigned any value, and is passed verbatim as the base parameter to both functions. Note that as the `elpclp800_setup` function will call these register access functions, they may not access any members of the DWC TRNG state structure or call any DWC TRNG SDK functions.

The readreg and writereg parameters are:

- `base` is the regbase member of the state structure,
- `offset` is the register index (in units of 32-bit words),
- (for `writereg`) val is the value to write.

Example usage

The following shows how the functions might be written for accessing the DWC TRNG through a hypothetical indirect interface similar to the DWC TRNG's own indirect access port:

```
void my_writereg(void *base_, unsigned offset, uint32_t val)
{
   uint32_t *base32 = base_;
   pdu_io_write32(&base32[MY_INDIRECT_ADDRESS], offset);
   pdu_io_write32(&base32[MY_INDIRECT_VALUE], val);
   pdu_io_write32(&base32[MY_INDIRECT_CONTROL], MY_INDIRECT_WRITE);
}

uint32_t my_readreg(void *base_, unsigned offset);
{
   uint32_t *base32 = base_;
   pdu_io_write32(&base32[MY_INDIRECT_ADDRESS], offset);
   pdu_io_write32(&base32[MY_INDIRECT_CONTROL], MY_INDIRECT_READ);
   return pdu_io_read32(&base32[MY_INDIRECT_VALUE]);
}

struct elpclp800_state *setup_hardware(uint32_t *my_indirect_base)
{
   struct elpclp800_state *clp800;

   clp800 = malloc(sizeof *clp800);
   if (clp800) {
         clp800->readreg = my_readreg;
         clp800->writereg = my_writereg;
         clp800->regbase = my_indirect_base;
         elpclp800_setup(clp800, NULL);
   }

   return clp800;
}
```

5.5 Device Configuration

After initializing the SDK with elpclp800_setup, the device can be configured using the functions in this section. Except for exceptional situations mentioned below, it is safe to perform engine reconfiguration concurrently with random number generation. However, concurrent threads must not attempt to simultaneously reconfigure the engine (by calling functions in this section) as the results are undefined.

5.5.1 Interrupts

If interrupts are being used, then the interrupt sources can be enabled or disabled by using the following functions:

```
void elpclp800_enable_irqs(struct elpclp800_state *clp800, uint32_t irq_mask);
void elpclp800_disable_irqs(struct elpclp800_state *clp800, uint32_t irq_mask);
```

where

- `clp800` points to an initialized state structure,
- `irq_mask` is the bitwise-OR of the interrupt sources to enable or disable.

Macros are defined to make it easier to set irq_mask. They correspond to the bits in the IE and ISTAT registers defined in the *DWC True Random Number Generator (TRNG) Databook*.

- `CLP800_IRQ_GLBL_EN_MASK`: global interrupt enable,
- `CLP800_IRQ_RQST_ALARM_MASK`: request-based reseed reminder interrupts,
- `CLP800_IRQ_AGE_ALARM_MASK`: age-based reseed reminder interrupts,
- `CLP800_IRQ_SEED_DONE_MASK`: random reseed completion interrupts,
- `CLP800_IRQ_RAND_RDY_MASK`: random number generation interrupts.

Note that if the global interrupt enable is not set, the device will not generate any interrupts.

5.5.2 Reseeding

A reseed process may be initiated using the following function:

```
int elpclp800_reseed(struct elpclp800_state *clp800, const void *nonce);
```

where

- `clp800` points to an initialized state structure,
- `nonce` points to the 32-byte seed to use, or NULL to initiate a random reseed.

Reseeds using a nonce complete immediately. Random reseeds complete asynchronously. The CLP800 remains fully functional during the reseed process and will continue to produce random numbers using the previous seed until the new one is ready. The availability of a new random `seed` is indicated by the `SEED_DONE` interrupt.

The function returns 0, if the reseed was successfully started. Defined errors are:

- `CRYPTO_MODULE_DISABLED` if a random reseed was requested but the hardware does not include the ring functionality.

Note that the hardware also has an external input pin to initiate random reseeds.

5.5.3 Secure Mode

The DWC TRNG can be run in either *secure* or *promiscuous* modes. Running in promiscuous mode allows host access to some internal state of the engine, for diagnostic purposes. The mode can be changed by calling the function

```
void elpclp800_set_secure(struct elpclp800_state *state, int secure);
```

where

- `clp800` points to an initialized state structure
- `secure` specifies the mode; 0 sets promiscuous mode, any other value sets secure mode.

Care must be taken when changing the mode of the engine, as this will zeroize all internal state and cancel any outstanding operations. The device will revert to the unseeded state, so it is normally necessary to explicitly reseed the engine after changing the mode.

5.5.4 Output Length Configuration

If the DWC TRNG was configured to support 256-bit output, then the output length may be changed at runtime between 128 and 256-bit modes. The maximum supported output size (in bytes) is stored in the output_len member of the state structure. Generally speaking, larger values perform better. Smaller sizes make individual random numbers generate faster. To change the output length of the engine, use the function:

```
int elpclp800_set_output_len(struct elpclp800_state *clp800, unsigned outlen);
```

where

- `clp800` points to an initialized state structure
- `outlen` specifies the output length in bytes (either 16 or 32).

The function returns 0 on success. Defined errors are

- `CRYPTO_MODULE_DISABLED` if the request was valid but not supported by the hardware

Care must be taken when reconfiguring the output length, as it only affects random number generations that were started after setting the output length. It is therefore recommended that the output length be set once at initialization and left unchanged thereafter. If the output length must be changed at runtime, it may be necessary to discard the next generated value.

Example usage

The following shows how the output length may be changed at runtime safely. If the application has concurrent threads, which may also access the device, it is necessary to block them from retrieving random numbers across the entire reconfiguration and subsequent polling loop (for example, by using a mutex if numbers are retrieved in polling mode, or by masking the device's interrupt contribution for interrupt mode).

```
int my_set_output_len(struct clp800_state *clp800, unsigned len)
{
   int ret;

   ret = elpclp800_set_output_len(clp800, len);
   if (ret == 0) {
         while (elpclp800_get_random(&priv->clp800, NULL) == CRYPTO_INPROGRESS)
               ;
   }

   return ret;
}
```

5.6 Mission Mode

Once the DWC TRNG is seeded, and the output length is (optionally) configured, it is ready to produce random output.

5.6.1 Generating Random Numbers

To start generating a new random value, write `CLP800_CMD_GEN_RAND` to the control register, for example

```
elpclp800_writereg(state, CLP800_CTRL, CLP800_CMD_GEN_RAND);
```

This operation completes asynchronously. The engine indicates that the operation has completed by asserting the `RAND_RDY` interrupt. Since the engine collects data in the background, to achieve highest throughput it should be requested to generate a new random number prior to when that value is actually needed.

5.6.2 Retrieving Random Numbers

Output from the DWC TRNG can be retrieved using the following function

```
int elpclp800_get_random(struct elpclp800_state *clp800, void *out);
```

where

- `clp800` points to an initialized state structure,
- `out` points to a buffer large enough[3] to hold the output from the engine. If this parameter is `NULL` the output will be discarded (but all other behavior, including status checks and clearing the `RAND_RDY` interrupt) will still be performed.

This function may be called from a `RAND_RDY` interrupt handler, but the engine is normally fast enough that polling mode is more efficient. Nevertheless, the DWC TRNG uses the interrupt status indicator to signal availability of output, so this function must be called *without* acknowledging the `RAND_RDY` interrupt. If a

[3] The buffer only needs to be as large as the configured output length, but it is safest to size the buffer to fit the largest possible output size (that is, 32 bytes). For convenience, an object-like macro is defined to expand to this maximum size: `ELPCLP800_MAXLEN`.

random value is successfully retrieved, the `RAND_RDY` interrupt flag will be cleared automatically by this function.

A return value of 0 indicates that there is no data available and the engine is not currently generating a new random number. If data is available, it is written to the provided output buffer and the number of bytes written (either 16 or 32) is returned. Otherwise, the defined errors are:

- `CRYPTO_NOT_INITIALIZED`, if the engine is not currently seeded. The engine must be seeded before random numbers can be generated.
- `CRYPTO_INPROGRESS`, if a random number is currently being generated.

When using polling mode with multiple concurrent threads, it is necessary to protect the hardware across the entire loop from simultaneous access by multiple threads (that is, by using a mutex). Such protections are normally not necessary in interrupt service routines as appropriate mutual exclusion is typically provided automatically by the hardware or operating system.

Example usage

The following example shows a very simple polling loop to retrieve a random value. It uses the return value of `elpclp800_get_random` to send a generate command, if the engine is idle.

```
int poll_random(struct elpclp800_state *clp800, void *out)
{
   int ret;

   do {
         ret = elpclp800_get_random(clp800, out);
         if (ret == 0) {
               /* Generate a new number */
               elpclp800_writereg(clp800, CLP800_CTRL, CLP800_CMD_GEN_RAND);
               ret = CRYPTO_INPROGRESS;
         }
   } while (ret == CRYPTO_INPROGRESS);

   return ret;
}
```

5.6.3 Reseed Reminder Alarms

The DWC TRNG may be configured to raise an interrupt, when a seed has been in use for a while, to remind software that it may be time to reseed the engine. The request-based reseed reminder may be set using the following function:

```
int elpclp800_set_request_reminder(struct elpclp800_state *clp800, unsigned long val);
```

where

- `clp800` points to an initialized state structure,
- `val` is the number of requests before the alarm is signalled, or 0 to disable the feature.

The function returns 0 on success. Note that the engine's internal counter has a resolution of 16 requests, so the specified request count will be rounded up to the nearest 16 requests.

The age-based reseed reminder may be set using the following function:

```
int elpclp800_set_age_reminder(struct elpclp800_state *clp800, unsigned long long val);
```

where

- `clp800` points to an initialized state structure,
- `val` is the number of clock cycles before the alarm is signalled, or 0 to disable the feature.

The function returns 0 on success. As with the request-based reminder, the engine's internal counter has a resolution of 2^{26} clock cycles, and the specified cycle count will be rounded up.

Changes to either reminder alarm do not take effect until the next time the engine is reseeded.

5.7 Diagnostics

When the engine is in promiscuous mode, it is possible to retrieve the seed material from the engine. This may be used to test the bit generation frontend. The seed material is retrieved using the function:

```
int elpclp800_get_seed(struct elpclp800_state *clp800, void *out);
```

where

- `clp800` points to an initialized state structure
- `out` points to a 32-byte buffer to store the output.

Returns 0 on success. Defined errors are:

- `CRYPTO_NOT_INITIALIZED` if the engine is unseeded.

If the engine is in secure mode, the result will always be all-bits zero.

6 DWC TRNG NIST SP800-90C

The DWC TRNG NIST SP800-90C is Synopsys' NIST SP800-90C and BSI AIS 20/31 compliant Random Number Generator. It generates random numbers that are intended to be statistically equivalent to a uniformly distributed random data stream. This chapter explains available driver SDK APIs to work with the DWC TRNG NIST SP800-90C.

- Section 6.1 explains the required API to operate the DWC TRNG NIST SP800-90C in the normal mode of operation.
- Section 6.2 provides the low-level API, solely dedicated to debugging and nonce seeding mode of operation. Using the functions in the low-level API needs extra care and detailed knowledge of the DWC TRNG NIST SP800-90C and NIST SP800-90a/b/c standards; hence, they are not recommended to be used in normal TRNG applications.

Note	This product is shipped as source code.

6.1 NIST SP800-90a/c Compliant API

This section introduces the NIST SP800-90a/c compliant API for the DWC TRNG NIST SP800-90C. This API is sufficient to operate the DWC TRNG NIST SP800-90C in the secure and mission mode of operation.

6.1.1 Structure

DWC TRNG NIST SP800-90C *state* struct is shown below:

```
typedef struct {
   uint32_t *base;

   /* Hardware features and build ID */
   struct {
      struct {
         elpclp890_drbg_arch drbg_arch;
         unsigned extra_ps_present,
                  secure_rst_state,
                  diag_level_trng3,
                  diag_level_stat_hlt,
                  diag_level_ns;
      } features;

      struct {
         unsigned stepping,
                  epn;
      } build_id;
   } config;

   /* status */
   struct {
      elpclp890_current_state current_state;
      unsigned nonce_mode,
               secure_mode,
               pred_resist;
      elpclp890_sec_strength sec_strength;
      unsigned pad_ps_addin;
      volatile unsigned alarm_code;
   } status;

   /* reminders and alarms */
   struct {
      unsigned long max_bits_per_req;
      unsigned long long max_req_per_seed;
      unsigned long bits_per_req_left;
      unsigned long long req_per_seed_left;
   } counters;
} elpclp890_state;
```

The `elpclp890_state` variable is passed to all functions in this API and contains the configuration, feature, status, and reminder information.

The AES engine inside the DWC TRNG NIST SP800-90C Hardware (HW) can be either a 128-bit or 256-bit AES. `drbg_arch` specifies, which AES architecture is instantiated inside the HW. The type, `elpclp890_drbg_arch`, is defined as shown below:

```
   typedef enum elpclp890_drbg_arch {
      AES128 = 0,
      AES256 = 1
   } elpclp890_drbg_arch;
```

The DWC TRNG NIST SP800-90C HW can be configured to accept personalization strings during the run-time in addition to the configured personalization string. If `extra_ps_present` is 1, it means that the HW can accept run-time personalization strings.

If the HW is configured to reset to the secure mode, `secure_rst_state` will be 1.

Diagnostic levels of the DWC TRNG NIST SP800-90C's internal noise source and health test circuit can be accessed from `diag_level_trng3`, `diag_level_n`, and `diag_level_stat_hlt`.

The `epn` member of `elpclp890_state` indicates the EPN number of the DWC TRNG NIST SP800-90C HW.

The status of the DWC TRNG NIST SP800-90C (combination of the HW and driver SDK) is stored in the status member. The `nonce_mode` indicates whether the DWC TRNG NIST SP800-90C is set to the self-seeding or nonce seeding mode. The `secure_mode` specifies whether the core is operating in the SECURE or PROMISCUOUS (diagnostic) mode of operation.

If the prediction resistance is available, `pred_resist` will be 1. The security strength (security algorithm) of the instantiated DRBG is determined by `sec_strength`. The DWC TRNG NIST SP800-90C provides security strengths of 128- and 256-bits. The type, `elpclp890_sec_strength`, is defined as shown below:

```
   typedef enum elpclp890_sec_strength {
      SEC_STRNT_AES128 = 0,
      SEC_STRNT_AES256 = 1
   } elpclp890_sec_strength;
```

`pad_ps_addin` is mainly meant to be used inside the driver's code.

When an alarm occurs, `alarm_code` determines the source of the alarm, which is read from the ALARM register.

The number of bits per generate request and the number of generate requests per seed have to be tracked and checked against the maximum allowed values, according to the NIST SP800-90a standard. The maximum values are programmed in `max_bits_per_req` and `max_req_per_seed`. Counters `bits_per_req_left` and `req_per_seed_left` keep track of the values.

This document assumes that the user has enough knowledge about the DWC TRNG NIST SP800-90C HW. For detailed information about the DWC TRNG NIST SP800-90C HW and its configurations, features and modes, refer to the *DWC TRNG NIST SP800-90C Databook*.

Note	The *state* struct described here is different from the *state_handle* input to DRBG functions in *NIST SP800-90C Databook*. NIST's *state_handle* is secret information of the DRBG and is kept secret inside the DWC TRNG NIST SP800-90C's HW.

6.1.2 Initialization

The DWC TRNG NIST SP800-90C driver SDK is initialized through the following function:

```
int elpclp890_init (elpclp890_state *state, uint32_t *base);
```

where:

state is a pointer to the SW state variable.

base is a pointer to the register base address.

This function will initialize the `state` variable using the base address pointed to by `base`. This function configures the DWC TRNG NIST SP800-90C in the secure and self-seeding mode of operation. It also turns on the prediction resistance and sets the security strength to the maximum possible value.

If the AES engine inside the DWC TRNG NIST SP800-90C HW is a 256-bit AES, the maximum security strength would be `SEC_STRNT_AES256` and if the AES engine is a 128-bit AES, the maximum security strength would be `SEC_STRNT_AES128`.

`max_bits_per_req` and `max_req_per_seed` are set to their default values specified by the following macros:

```
#define CLP890_DFLT_MAX_BITS_PER_REQ (1ul<<19)
#define CLP890_DFLT_MAX_REQ_PER_SEED (1ull<<48)
```

NIST SP800-90a sets the limit of 2^{19} for the maximum number of bits per generate request, and 2^{48} for the maximum number of generate requests per seed.

6.1.3 Instantiate

The following function implements the NIST SP800-90A's ***Instantiate_function***, which instantiates the DRBG inside the DWC TRNG NIST SP800-90C:

```
int elpclp890_instantiate (elpclp890_state *state,
                           int req_sec_strength, int pred_resist,
                           void *personal_str);
```

where:

state is a pointer to the SW state variable.

req_sec_strength is the requested security strength to instantiate the DRBG.

pred_resist determines the prediction resistance availability.

personal_str is a pointer to the personalization string.

This function follows these steps:

1. If the DRBG is already instantiated, the function returns error.
2. Sets the security strength of the DRBG according to the *req_sec_strength* input. Error will be returned if *req_sec_strength* is invalid.

A valid security strength has to be an integer more than 0 and less than or equal to 256. The function choses one of `SEC_STRNT_AES128` or `SEC_STRNT_AES256` as follows:

- 0 < `req_sec_strength` <= 128 → security strength = `SEC_STRNT_AES128`
- 128 < `req_sec_strength` <= 256 → security strength = `SEC_STRNT_AES256`
- else → Invalid security strength

3. If the `personal_str` pointer is NULL, it means that the extra personalization string does not exist. If the pointer is not NULL, but the HW is not configured to accept an extra personalization string at run-time, the function returns error.
4. Initiates an entropy generation operation and waits until the operation is completed. If entropy generation is not successful, the function returns with a FATAL error.

Note	In the nonce mode, the user must seed the DRBG by calling `elpclp890_get_entropy_input` prior to the instantiate request. In the noise mode, the instantiate function takes care of this function call. For more information on how to operate the DWC TRNG NIST SP800-90C in the nonce mode of operation, refer to the *NIST SP800-90C Databook.*

5. If the HW is configured to accept extra personalization string and the `personal_str` pointer is not NULL, the function loads the **NPA_DATAx** registers with the data pointed to by `personal_str`. If the HW is configured to accept extra personalization string, but, the pointer is NULL, the function will load the **NPA_DATAx** registers with zeros.

Note	The length of the personalization string depends on the requested security strength. If `SEC_SRTNT_AES128` is chosen, the string must be 256-bits and if `SEC_STRNT_AES256` is chosen, the string must be 384-bits. It is the user's responsibility to provide a full 256/384 bits of data for the personalization string.

6. Initiates the state creation command.
7. Waits for the command to finish.
8. Sets the prediction resistance according to `pred_resist`. The prediction resistance does not affect the instantiation process, because this function always gets fresh entropy from the entropy source inside the HW; hence, prediction resistance is always present. However, according to the NIST SP800-90a, if the prediction resistance is set to 0 at this stage, a change to 1 is not allowed as long as the current DRBG instantiation exists. The prediction resistance can change from 1 to 0 at any stage.

Note	Prediction resistance always exists in the self-seeding mode of operation. In the nonce mode of operation, it is the user's responsibility to provide fresh entropy, if prediction resistance is required.

9. Resets all the reminder counters.

6.1.4 Uninstantiate

The following function implements the NIST SP800-90A’s ***Uninstantiate_function***, which removes the DRBG instantiation and cleans up all the secret information inside the core (zeroize operation):

```
int elpclp890_uninstantiate (elpclp890_state *state);
```

where:

state is a pointer to the SW state variable.

If the DRBG has not been instantiated, yet, this function will return the `CRYPTO_NOT_INSTANTIATED` flag, however, the function will still zeroize the core. Hence, this function can be used to zeroize the core at any stage of operation.

6.1.5 Reseed

Reseed will insert additional entropy into the current state of the DRBG. The following function implements the NIST SP800-90A's ***Reseed_function***, which performs the reseed operation:

```
int elpclp890_reseed (elpclp890_state *state, int pred_resist, void *addin_str);
```

where:

state is a pointer to the SW state variable.

pred_resist determines the prediction resistance availability.

addin_str is a pointer to the additional input string.

This function follows these steps:

1. If the DRBG's state has not been instantiated, returns error.
2. Sets the prediction resistance. If the prediction resistance in the state is 0, but `pred_resist` is 1, returns error.

Note	Prediction resistance always exists in the self-seeding mode. In the nonce mode of operation, it is the user's responsibility to provide fresh entropy, if the prediction resistance is required.

3. Initiates an entropy generation operation and waits until the operation is complete. If entropy generation is not successful, the function returns with a FATAL error.

Note	In the nonce mode, the user must seed the DRBG by calling `elpclp890_get_entropy_input` prior to the reseed request. In the noise mode, the reseed function takes care of this function call. For more information about the nonce mode of operations, refer to the Section 6.2.8.

4. If the `addin_str` pointer is not NULL, it means that the additional input string is present. If the additional input string is present, the function sets the **ADDIN_PRESENT** field of the **MODE** register to 1, and loads the **NPA_DATAx** registers with the data pointed to by `addin_str`.

Note	The length of the addition input string depends on the requested security strength. If `SEC_SRTNT_AES128` is chosen, the string must be 256-bits and if `SEC_STRNT_AES256` is chosen, the string must be 384-bits. It is the user's responsibility to provide a full 256/384 bits of data for the additional input string.

5. Initiates the reseed operation.
6. Waits for the command to finish.
7. Resets all the reminder counters.

6.1.6 Generate

The following function implements the NIST SP800-90A's ***Generate_function***, which generates random number output:

```
int elpclp890_generate (elpclp890_state *state,
                        void *random_bits, unsigned long req_num_bytes,
                        int req_sec_strength, int pred_resist,
                        void *addin_str);
```

where:

state is a pointer to the SW state variable.

random_bits is a pointer to the output buffer.

req_num_bytes is the requested number of random bytes to generate.

req_sec_strength is the requested security strength.

pred_resist determines the prediction resistance availability.

addin_str is a pointer to the additional input string.

This function follows these steps:

1. If the DRBG's state has not been instantiated, returns error.
2. If the requested number of bits, determined by 8×`req_num_bytes`, is more than the maximum number of bits per request as specified by `max_bits_per_req` in `state`, returns error.
3. If the requested security strength, indicated by `req_sec_strength`, is not valid or if it is more than the security strength that the DRBG is instantiated with, returns error. For information about valid security strength, refer to the Section 6.1.3.
4. Sets the prediction resistance. If prediction resistance in the state is 0, but, `pred_resist` is 1, returns error.
5. Sets a `reseed_required` flag to 0.
6. If prediction resistance is required or `reseed_required` is 1, calls `elpclp890_reseed`. Also, sets the `addin_str` pointer to NULL, because according to NIST SP800-90a additional input should not be used in the generate process, if a reseed is executed. The additional input would only be used in the Reseed function.

Note	If the user sets `pred_resist` to 0 and calls the reseed function prior to the generate function to provide prediction resistance, they must pass NULL to the generate function as the additional input string. However, in that case, if a reseed becomes required at the end, additional input won't be available for the reseed operation. Because of this, and the NIST SP800-90a requirement that individual reseed and generate calls by the consumer cannot be qualified to provide prediction resistance, it is strongly recommended to avoid using the `elpclp890_reseed` function and use `elpclp890_generate` together with the desired `pred_resist`.

7. If the `addin_str` pointer is not NULL, or if the function does not set the `addin_str` to NULL (as the consequence of a reseed operation), it means that the additional input string is present and will be used in the generate procedure to add to the entropy.

Note	Length of the addition input string depends on the requested security strength. If `SEC_SRTNT_AES128` is chosen, the string must be 256-bits and if `SEC_STRNT_AES256` is chosen, the string must be 384-bits. It is the user's responsibility to provide a full 256/384-bits of data for the additional input string.

8. Generates "`8×req_num_bytes`" random bits and store them in `random_bits` string.
9. If the seed's lifetime or the maximum number of request is reached, set the `reseed_required` to 1 and jump back to Step 6.

6.1.7 Example Code

This section provides some examples on how to use the functions in the NIST compliant API of the DWC TRNG NIST SP800-90C Driver SDK package.

Following variables are used in the examples:

```
elpclp890_state *clp890_state; // state variable
uint32_t addin[12];            // additional input
uint32_t ps[12];               // personalization string
char *out;                     // pointer to store the output random number
uint32_t *base_addr;           // pointer to the register base address
/* addin and ps are assumed to be initialized */
/* out is assumed to be allocated */
/* base_addr is assumed to be set properly to the register base address */
```

Driver SDK has to be initialized before any function can be called:

```
if (elpclp890_init(clp890_state, base_addr) < 0) { return -1; }
```

6.1.7.1 Prediction Resistance Not Available, No Reseed

In this example, DRBG is instantiated with 128-bits of security strength. Prediction resistance is not requested and generate requests are issued without any reseed operation in between.

```
// instantiate, sec_strength=128, pred_resist=0
if (elpclp890_instantiate(clp890_state, 128, 0, ps) < 0)          { return -1; }
// generate 10 bytes of random number, sec_strength=128, pred_resist=0
if (elpclp890_generate(clp890_state, out, 10, 128, 0, addin) < 0) { return -1; }
// generate 512 bytes of random number, sec_strength=128, pred_resist=0
if (elpclp890_generate(clp890_state, out, 512, 128, 0, addin) < 0){ return -1; }
// generate 41 bytes of random number, sec_strength=128, pred_resist=0
if (elpclp890_generate(clp890_state, out, 41, 128, 0, addin) < 0) { return -1; }
// uninstantiate
if (((err = elpclp890_uninstantiate(clp890_state)) < 0) &&
     (err != CRYPTO_NOT_INSTANTIATED))                            { return -1; }
```

`CRYPTO_NOT_INSTANTIATED` is returned from `elpclp890_uninstantiate` when the DRBG is not instantiated, but the uninstantiate function is called. Uninstantiate function still performs the zeroize operation when the DRBG is not instantiated, hence, `CRYPTO_NOT_INSTANTIATED` is an informational flag and not an error flag.

6.1.7.2 Prediction Resistance Not Available, With Reseed

In this example, DRBG is instantiated with 256-bits of security strength. Prediction resistance is not requested. Reseed is performed, occasionally.

```
// instantiate, sec_strength=256, pred_resist=0
if (elpclp890_instantiate(clp890_state, 256, 0, ps) < 0)          { return -1; }
// reseed, pred_resist=0
if (elpclp890_reseed(clp890_state, 0, addin) < 0)                 { return -1; }
// generate 64 bytes of random number, sec_strength=256, pred_resist=0
if (elpclp890_generate(clp890_state, out, 64, 256, 0, addin) < 0) { return -1; }
// generate 64 bytes of random number, sec_strength=256, pred_resist=0
if (elpclp890_generate(clp890_state, out, 64, 256, 0, addin) < 0) { return -1; }
// generate 64 bytes of random number, sec_strength=256, pred_resist=0
if (elpclp890_generate(clp890_state, out, 64, 256, 0, addin) < 0) { return -1; }
// reseed, pred_resist=0
if (elpclp890_reseed(clp890_state, 0, addin) < 0) { return -1; }
// generate 64 bytes of random number, sec_strength=256, pred_resist=0
if (elpclp890_generate(clp890_state, out, 64, 256, 0, addin) < 0) { return -1; }
// uninstantiate
if (((err = elpclp890_uninstantiate(clp890_state)) < 0) &&
     (err != CRYPTO_NOT_INSTANTIATED))                            { return -1; }
```

6.1.7.3 Prediction Resistance is Available, No Reseed

In this example, DRBG is instantiated with 128-bits of security strength. Prediction resistance is requested and generate requests are issued without any reseed operation in between.

```
// instantiate, sec_strength=128, pred_resist=0
if (elpclp890_instantiate(clp890_state, 128, 1, ps) < 0)           { return -1; }
// generate 120 bytes of random number, sec_strength=128, pred_resist=1
if (elpclp890_generate(clp890_state, out, 120, 128, 1, addin) < 0) { return -1; }
// generate 120 bytes of random number, sec_strength=128, pred_resist=1
if (elpclp890_generate(clp890_state, out, 120, 128, 1, addin) < 0) { return -1; }
// uninstantiate
if (((err = elpclp890_uninstantiate(clp890_state)) < 0) &&
     (err != CRYPTO_NOT_INSTANTIATED))                             { return -1; }
```

Since prediction resistance is requested, a reseed operation gets executed before each generate function.

6.1.7.4 Zeroize

In this example, DRBG is not instantiated. An uninstantiate function is called to zeroize the core.

```
if ((err = elpclp890_uninstantiate(clp890)) &&
    (err != CRYPTO_NOT_INSTANTIATED))           { return -1; }
```

6.1.7.5 Illegal Sequences

This example presents few examples of illegal sequences, which will cause error.

DRBG has to be instantiated before generate or reseed functions can be called. In the following two example codes, generate and reseed functions are called before the DRBG is instantiated. Both function calls will return with error.

```
// generate before the DRBG is instantiated
elpclp890_generate(clp890, out, 120, 128, 0, addin);

// reseed before the DRBG is instantiated
elpclp890_reseed(clp890, 0, addin);
```

DWC TRNG NIST SP800-90C HW state expects a reseed operation to be followed by a generate process (for detailed state diagram, refer to the *DWC TRNG NIST SP800-90C Databook*). In the example, below, second reseed will return with an error. Also, HW generates an alarm.

```
// two consecutive reseed function calls
if (elpclp890_instantiate(clp890, 128, 0, ps) < 0) { return -1; }
if (elpclp890_reseed(clp890, 0, addin) < 0) { return -1; }
elpclp890_reseed(clp890, 0, addin);
```

6.2 Low Level API

Functions introduced in this section are not meant to be used by users for the normal mode of operation. They are mainly used for debugging purposes or the nonce seeding mode of operation (where the user wants to provide the nonce instead of using the internal entropy source of the DWC TRNG NIST SP800-90C). Using the functions in the low-level API needs extra care and detailed knowledge of the DWC TRNG NIST SP800-90C and NIST SP800-90a/b/c standards; hence, they are not recommended to be used in normal TRNG applications.

6.2.1 Reminders

Reminders introduced in the Section 6.1.1 and initialized by elpclp890_init function can change using the following functions:

```
int elpclp890_set_reminder_max_bits_per_req (elpclp890_state *state,
                                             unsigned long max_bits_per_req);
int elpclp890_set_reminder_max_req_per_seed (elpclp890_state *state,
                                             unsigned long long max_req_per_seed);
```

where:

state is a pointer to the SW state variable.

max_bits_per_req is the requested maximum bits per request.

max_req_per_seed is the requested maximum generate requests per seed.

If these functions are called when the DRBG is instantiated, an error will be returned. If the requested maximum values are more than the standard limits (as mentioned in the Section 6.1.2), these functions will return error.

6.2.2 Set the Secure Mode

The following function puts the DWC TRNG NIST SP800-90C in either the *SECURE* or *PROMISCUOUS* mode.

```
int elpclp890_set_secure_mode (elpclp890_state *state, int secure_mode);
```

where:

state is a pointer to the SW state variable.
secure_mode determines the security mode.

A value of 1 for `secure_mode` puts the core in the SECURE mode and a value of 0 puts the core in the PROMISCUOUS mode. Any change to the secure mode of the DWC TRNG NIST SP800-90C will result in a complete zeroize, and will set the seeding mode to self-seeding. A zeroize will not destroy the programmed mode and ALARM register value. It keeps the programmed mode to avoid re-programming. It also, maintains the ALARM register value, so that the user can read the value to understand the reason of the occurred alarm.

6.2.3 Set the Seeding Mode (Nonce versus Self)

```
int elpclp890_set_nonce_mode (elpclp890_state *state, int nonce_mode);
```

where:

state is a pointer to the SW state variable.

nonce_mode determines the seeding mode.

This function can be used to change the seeding mode of the DWC TRNG NIST SP800-90C. A value of 1 for `nonce_mode` will put the DWC TRNG NIST SP800-90C in the nonce seeding mode, which means that the seed will be provided by the user, unlike the noise or self-seeding mode (normal mode of operation) in which the seed is generated by the internal entropy source.

Any transition to or from the nonce mode will zeroize the DWC TRNG NIST SP800-90C.

6.2.4 Set the Security Strength

```
int elpclp890_set_sec_strength (elpclp890_state *state, int req_sec_strength);
```

where:

state is a pointer to the SW state variable.

req_sec_strength is the requested security strength.

This function sets the security strength of the DRBG instance. The function chooses one of `SEC_STRNT_AES128` or `SEC_STRNT_AES256` as explained in the Section 6.1.3.

If the DRBG is instantiated, a new security strength change request with greater security strength will return error.

6.2.5 Set the ADDIN_PRESENT Field

```
int elpclp890_set_addin_present (elpclp890_state *state, int addin_present);
```

where:

state is a pointer to the SW state variable.

addin_present determines the availability of the additional input.

This function sets the **ADDIN_PRESENT** field of the **MODE** register according to the `addin_present` input.

6.2.6 Set the PRED_RESIST Field

```
int elpclp890_set_pred_resist (elpclp890_state *state, int pred_resist);
```

where:

state is a pointer to the SW state variable.

addin_present determines the availability of the prediction resistance.

This function sets the **PRED_RESIST** field of the **MODE** register according to the `pred_resist` input. If the DRBG is instantiated with prediction resistance of 0, and a change to the prediction resistance of 1 is requested, the function will return error.

6.2.7 Load the NPA_DATAx Register

The following function can be used to load the additional input or personalization string into the **NPA_DATAx** registers.

```
int elpclp890_load_ps_addin (elpclp890_state *state, void *input_str);
```

where:

state is a pointer to the SW state variable.

input_str is a pointer to the input string (personalization or additional input string).

The function loads the proper number of bits (256 or 384) according to the security strength stored in the state handle.

6.2.8 Get Entropy

If the DWC TRNG NIST SP800-90C is in the nonce mode, entropy must be provided by the user; otherwise, the entropy will be generated by the internal entropy source of the DWC TRNG NIST SP800-90C.

The following function provides the entropy:

```
int elpclp890_get_entropy_input (elpclp890_state *state,
                                 void *input_nonce, int nonce_operation);
```

where:

state is a pointer to the SW state variable.

input_nonce is a pointer to the input nonce string.

nonce_operation determines the nonce seeding method (direct versus through derivation function).

The self-seeding mode of this function implements the NIST SP800-90A's ***Get_entropy_input***. Although the nonce mode of this function provides entropy rather than generating or getting entropy, the nonce mode is incorporated inside the same function as the self-seeding mode, because they both share the same functionality, which is generating seed for the DRBG.

Inputs 2 and 3 are only used when the core is in the nonce mode. In the noise mode, calling the function will initiate a seeding command. Depending on the programmed security strength, a 256-bit or 384-bit seed will be generated.

In the nonce mode, the DWC TRNG NIST SP800-90C can be seeded either through 2 or 3 blocks of 512-bit nonce values, which are passed to the internal derivation function to increase the entropy, or the DWC TRNG NIST SP800-90C can be seeded by a 256 or 384-bit nonce written directly into the **SEEDx** registers. Passing a value of 1 to `nonce_operation`, selects the former scenario and a value of 0 selects the latter. The `input_nonce` pointer must point to a memory location with a sufficient number of initialized bits. Table 6-1 shows the required number of bits depending on the `nonce_operation` and the security strength values.

Table 6-1 Bit Length Requirement for User Provided Nonce

nonce_operation	Security Strength	Bit Length Requirement
1 (using the Derivation Function)	**SEC_STRNT_AES128**	2x512 = 1024
1 (using the Derivation Function)	**SEC_STRNT_AES256**	3x512 = 1536
0 (loading the seed into SEEDx)	**SEC_STRNT_AES128**	256
0 (loading the seed into SEEDx)	**SEC_STRNT_AES256**	384

Generated entropy is secret information held securely within the HW and remains inaccessible to the user, unless the HW core is in the PROMISCUOUS mode.

6.2.9 Generate Function

The generate function in DWC TRNG NIST SP800-90C HW is broken down into three steps: Refresh_Addin, Gen_Random, and Advance_State. `elpclp890_generate` described in the Section 6.1.6 incorporates all these steps and some extra checks into one simple function.

6.2.9.1 Generate Part 1: Referesh_Addin

The first step of the generate function is Refresh_Addin, where the additional input string is used to add to the HW state entropy. The following function performs the Refresh_Addin:

```
int elpclp890_refresh_addin (elpclp890_state *state, void *addin_str);
```

where:

state is a pointer to the SW state variable.

addin_str is a pointer to the additional input string.

This function calls `elpclp890_set_addin_present` to set the **ADDIN_PRESENT** field of the **MODE** register to 1. Then, the function loads the additional input provided by `addin_str` pointer into the **NPA_DATAx** registers by calling the `elpclp890_load_ps_addin`. At the end, the function issues a **Refresh_Addin** command to the HW.

If the `addin_str` pointer is NULL, the function will return error.

6.2.9.2 Generate Part 2: Gen_Random

The second step of the generate process is Gen_Random, which generates the requested number of bits. The function is:

```
int elpclp890_gen_random (elpclp890_state *state,
                          void *random_bits, int req_num_bytes);
```

where:

state is a pointer to the SW state variable.

random_bits is a pointer to the generated random number.

req_num_blks is the requested number of random bytes to generate.

This function issues the Gen_Random command to the HW as many times as necessary to generate `req_num_bytes` number of bytes. If the requested number of bits (that is, 8×`req_num_bytes`) is more than the maximum value specified by `max_bits_per_req`, the function will return error.

If `req_num_bytes` is not a multiple of 16, which is the number of bytes generated by each issuance of the HW Gen_Random command, bits generated by the last Gen_Random command will be used, partially. `elpclp890_gen_random` discards unused bits from the MSB side.

Random bits will be returned in `random_bits`.

6.2.9.3 Generate Part 3: Advance_State

The last step of the generate process advances the state of the DRBG. The function is:

```
int elpclp890_advance_state (elpclp890_state *state);
```

where:

state is a pointer to the SW state variable.

This function issues the Advance_State command to the HW. Then, the counter for the number of generate requests per seed gets updated. The counter must be checked every time before starting the generate process and a reseed must be issued, if the limit is reached. This check is incorporated inside `elpclp890_generate` function discussed in the Section 6.1.6.

6.2.10 Known Answer Test (KAT)

The DWC TRNG NIST SP800-90C can perform a KAT on the DRBG and the derivation function inside the entropy source. There are two different vectors available to do the KAT. The following function can be used to perform the KAT:

```
int elpclp890_kat (elpclp890_state *state, int kat_sel, int kat_vec);
```

where:

state is a pointer to the SW state variable.

kat_sel selects the circuit to perform the KAT on.

kat_vec selects the KAT vector.

The `kat_sel` input selects whether the KAT should be performed on the DRBG or the derivation function. The `kat_vec` input chooses the KAT vector. Selections are done by writing the values to the MODE register. If the KAT fails, the function returns error.

The following function can be used to perform a full KAT with all four combinations of the `kat_sel` and `kat_vec`:

```
int elpclp890_full_kat (elpclp890_state *state);
```

where:

state is a pointer to the SW state variable.

If any of the KAT fails, the function returns error.

6.2.11 Zeroize

The following function can be used to do a HW zeroize, which will erase all the secure information inside the HW and removes the DRBG instance if existed:

```
int elpclp890_zeroize (elpclp890_state *state);
```

where:

state is a pointer to the SW state variable.

6.2.12 Example code

This section provides an example code that can be used in validation process of the DWC TRNG NIST SP800-90C's DRBG. The code shows how to run NIST DRBG vectors on the DWC TRNG NIST SP800-90C's DRBG to validate the functionality. NIST provides vectors in different scenarios: prediction resistance not available (PR_FALSE) in which it does manual reseed, prediction resistance available (PR_TRUE) in which reseed happens as part of the generate process, and no prediction resistance and no reseed (NO_RESEED) in which no reseed happens.

In this example, a vector from PR_TRUE vectors file is picked, which has both of the personalization string and additional input string:

```
[AES-256 no df]
[PredictionResistance = True]
[EntropyInputLen = 384]
[NonceLen = 0]
[PersonalizationStringLen = 384]
[AdditionalInputLen = 384]
[ReturnedBitsLen = 512]

COUNT = 0
EntropyInput =
c54805274bde00aa5289e0513579019707666d2fa7a1c8908865891c87c0c652335a4d3cc415bc30742b164
647f8820f
Nonce =
PersonalizationString =
d63fb5afa2101fa4b8a6c3b89d9c250ac728fc1ddad0e7585b5d54728ed20c2f940e89155596e3b963635b6
d6088164b
AdditionalInput =
744bfae3c23a5cc9a3b373b6c50795068d35eb8a339746ac810d16f864e880061082edf9d2687c211960aa8
3400f85f9
EntropyInputPR =
b2ad31d1f20dcf30dd526ec9156c07f270216bdb59197325bab180675929888ab699c54fb21819b7d921d63
46bff2f7f
AdditionalInput =
ad55c682962aa4fe9ebc227c9402e79b0aa7874844d33eaee7e2d15baf81d9d33936e4d93f28ad109657b51
2aee115a5
EntropyInputPR =
eca449048d26fd38f8ca435237dce66eadec7069ee5dd0b70084b819a711c0820a7556bbd0ae20f06e51692
78b593b71
ReturnedBits =
f08fdfc1775b6feb5a4177110bf29d7c3ab715dfdc4b27200359288c0624bd5c1028acc9914d88a82b09f5e
aafdc3bca8547b98481df39b86504314221cbdc3c
```

Required variables are:

```
elpclp890_state *clp890_state; // state variable
char *out;                     // pointer to store the output random number
uint32_t *base_addr;           // pointer to the register base address
char buf_seed1[96]  =
"c54805274bde00aa5289e0513579019707666d2fa7a1c8908865891c87c0c652335a4d3cc415bc30742b16
4647f8820f";
char buf_ps1[96]    =
"d63fb5afa2101fa4b8a6c3b89d9c250ac728fc1ddad0e7585b5d54728ed20c2f940e89155596e3b963635b
6d6088164b";
```

```
char buf_addin1[96] =
"744bfae3c23a5cc9a3b373b6c50795068d35eb8a339746ac810d16f864e880061082edf9d2687c211960aa
83400f85f9";
char buf_seed2[96]  =
"b2ad31d1f20dcf30dd526ec9156c07f270216bdb59197325bab180675929888ab699c54fb21819b7d921d6
346bff2f7f";
char buf_addin2[96] =
"ad55c682962aa4fe9ebc227c9402e79b0aa7874844d33eaee7e2d15baf81d9d33936e4d93f28ad109657b5
12aee115a5";
char buf_seed3[96]  =
"eca449048d26fd38f8ca435237dce66eadec7069ee5dd0b70084b819a711c0820a7556bbd0ae20f06e5169
278b593b71";
uint32_t tmp[12];
/* out is assumed to be allocated */
/* base_addr is assumed to be set properly to the register base address */
```

Driver SDK has to be initialized before any function can be called:

```
if (elpclp890_init(clp890_state, base_addr) < 0) { return -1; }
```

Code to run and generate the random output bit is as follows:

```
// set the core in the nonce mode
if (elpclp890_set_nonce_mode(&priv->elpclp890, 1) < 0) { return -1; }

// ----- Instantiate
// covert the character string to bit stream
str_to_384_bit(buf_seed1, tmp);
// seed the DRBG by loading the seed directly into the SEEDx registers
if (elpclp890_get_entropy_input(&priv->elpclp890, tmp, 0) < 0) { return -1; }
// covert the character string to bit stream
str_to_384_bit(buf_ps1, tmp);
// instantiate the DRBG, , sec_strength=256, pred_resist=1
if (elpclp890_instantiate(&priv->elpclp890, 256, 1, tmp) < 0) { return -1; }

// ----- First Generate
// covert the character string to bit stream
str_to_384_bit(buf_seed2, tmp);
// seed the DRBG by loading the seed directly into the SEEDx registers
// this seed will be used for the prediction resistance reseed
if (elpclp890_get_entropy_input(&priv->elpclp890, tmp, 0) < 0) { return -1; }
// covert the character string to bit stream
str_to_384_bit(buf_addin1, tmp);
// generate 64 bytes of random number, sec_strength=256, pred_resist=1
if (elpclp890_generate(&priv->elpclp890, out, 64, 256, 1, tmp) < 0) { return -1; }

// ----- Second Generate
// covert the character string to bit stream
str_to_384_bit(buf_seed3, tmp);
// seed the DRBG by loading the seed directly into the SEEDx registers
// this seed will be used for the prediction resistance reseed
if (elpclp890_get_entropy_input(&priv->elpclp890, tmp, 0) < 0) { return -1; }
// covert the character string to bit stream
str_to_384_bit(buf_addin2, tmp);
// generate 64 bytes of random number, sec_strength=256, pred_resist=1
if (elpclp890_generate(&priv->elpclp890, out, 64, 256, 1, tmp) < 0) { return -1; }
```

at this point, must contain the 512-bit stream:

```
f08fdfc1775b6feb5a4177110bf29d7c3ab715dfdc4b27200359288c0624bd5c1028acc9914d88a82b09f5e
aafdc3bca8547b98481df39b86504314221cbdc3c
```

The `str_to_384_bit` function converts a string to a hexadecimal bit stream. An implementation of the function is as shown below:

```
static void str_to_384_bit (char *buf, uint32_t *out) {
   char foo[9];
   int i;

   foo[8] = 0;
   for (i = 0; i < 12; i++) {
      memcpy(foo, buf + i*8, 8);
      kstrtouint(foo, 16, out+11-i);
   }
}
```